AFRL-SR-AR-TR-05-
0067

REPORT DOCUMENTATION PAGE

The public reporting burden for this collection of information is estimated to average 1 hour per response, including gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Servic 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwiths penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*

2. REPORT TYPE Final

3. DATES COVERED *(From - To)* 12 December 2000 - 30 November 2004

4. TITLE AND SUBTITLE
Self-Organizing and Autonomous Learning Agents and Systems

5a. CONTRACT NUMBER

5b. GRANT NUMBER F49620-01-1-0020

5c. PROGRAM ELEMENT NUMBER

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

6. AUTHOR(S)
Wei-Min Shen

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Department of Polymorphic Robotics Laboratory
University of Southern California
4676 Admiralty Way
Marina del Ray, CA 90292-6695

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Office of Scientific Research
4015 Wilson Blvd
Mail Room 713
Arlington, VA 22203 NM

10. SPONSOR/MONITOR'S ACRONYM(S) AFOSR

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution Statement A. Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

During the period of 9/1/2003 through 8/31/2004 we have developed solutions for two critical problems faced by self-reconfigurable systems: (1) Autonomous discovery and response to unexpected topology changes; (2) A new distributed functional language called DH2 for programming of self-reconfigurable systems using hormone-inspired computational methods. These results are published by Dr Wei-Min Shen and his students Behnam Salemi, Maks Krivokon, and Michael Rubenstein.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Wei-Min Shen
U	U	U	UU		19b. TELEPHONE NUMBER *(Include area code)* 310-448-8710

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

To USAF, Air Force Office of Scientific Research
Program Manager: Dr. Robert Herklotz

Self-Organizing and Autonomous Learning Agents and Systems (Final Report)

Award Number: F49620-01-1-0020
Period of Performance: 12/01/2000 – 11/30/2004

PREPARED BY:

Dr. Wei-Min Shen
Director of Polymorphic Robotics Laboratory
Information Sciences Institute
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292-6695
Phone: 310-448-8710
Fax: 310-822-0751
shen@isi.edu
http://www.isi.edu/robots

20050309 075

Table of Contents

1 Objectives 3
2 Status of Effort 4
3 Accomplishments and New Findings 5
  3.1 2000-2001 Period 5
    3.1.1 The Construction of Basic Autonomous Agents 5
    3.1.2 The Digital Hormone Model (DH-Model) for Self-Organization 11
    3.1.3 Self-Reconfigurable Robots and CONRO Modules 16
    3.1.4 Applications of DH-Model to Metamorphic Robots 18
  3.2 2001-2002 Period 22
    3.2.1 Adaptive Communication 24
    3.2.2 Hormone-Inspired Distributed Control 26
    3.2.3 Experimental Results 30
  3.3 2002-2003 Period 32
    3.3.1 Distributed Task Negotiation 32
    3.3.2 Distributed Behavior Selection 40
  3.4 2003-2004 Period 48
    3.4.1 Autonomous Discovery and Response to Unexpected Topology Changes 48
    3.4.2 DH2: Distributed Functional Language for Self-Reconfigurable Systems 54
4 Personnel Supported 63
5 Publications 63
6 Interactions and Transitions 65
7 New Discoveries, Inventions, or Patent Disclosures 66
8 Honors and Awards 66

1 Objectives

The main objective for the Self-Organizing and Autonomous Learning Agents (SOALA) project is to develop synthetic and physical autonomous agents that can learn to perform complex tasks and self-reconfigure their shape, size and configurations while executing missions in dynamic, uncertain, and even unforeseen environments. The agents must autonomously learn a model of its own actions from the environment, and then reconfigure their physical or logical relations to improve their performance in the environment.

A self-reconfigurable system is typically made from a network of homogeneous or heterogeneous *reconfigurable modules/agents* that can autonomously change their physical or logical connections and rearrange their configurations. Self-reconfiguration provides some great opportunities for both sciences and applications. In sciences, principles of self-reconfiguration could help us deepen our understanding how natural systems are evolved and adaptive to their environment, these understandings would suggest new designs for new engineering systems such as smart materials or self-assembly systems. In applications, self-reconfiguration provides a critical capability for solving many real-world problems. In homeland security, self-reconfiguration will allow unmanned ground or air vehicles to restructure and repair their organization in unexpected situations. In information systems, it will enable software agents to adjust their relationship to gather and deliver critical information in a timely fashion. In search-rescue or inspect-repair applications, self-reconfigurable robots can maneuver in tight spaces that are hard to reach by humans or conventional robots. In such an application, a self-reconfigurable robot could become a ball to roll down a slope, slither down between stones as a "snake" to locate a person or artifact, morph smoothly into a "crab" and climb over rubble, and then transform a leg into a gripper to grasp and carry objects. A snake robot can be cut into pieces, yet each piece will continue function for the mission. An octopus robot can lose legs, yet it can "grow" new legs by rearranging modules from other legs. A self-reconfigurable system can also disassemble and distribute it self into many small and agile units for certain tasks, and then re-assemble the small units back into the original single configuration. In deepwater operations, a self-reconfigurable robot may become an eel for swimming in open water, change into an octopus for grasping objects, and then "spread" itself into many small but agile units to monitor large areas. In space environments, these robots can be applied to self-assembly of large structures in a incremental and economic fashion. All these scenarios are not only for robots but also for any system that has a great number of agents and that must dynamically reconfigure the system organizations.

Our approaches to self-reconfigurable systems are twofold. On the theoretical side, we attempt to understand the phenomena of self-reconfiguration in nature, and then develop a general theory for self-reconfigurable systems. On the engineering side, we design and build novel mechanisms and software systems for self-reconfigurable robots and agents, and apply these them to complex tasks in space, deepwater, or any other environment that requires self-reconfigurable capabilities.

2 Status of Effort

During the period of 12/01/2000 through 8/31/2001, we had built a set of basic autonomous agents in a general and flexible 3D simulation environment called Morfit (http://www.3dstate.com/), developed a learning methodology called Object-Centered Explorative Modeling (OCEM) for learning agents, and formulated a biologically-inspired approach called "digital hormones" for self-organization among agents. The Morfit 3D simulation environment allows SOALA agents to see the environment through simulated cameras and range detectors, to maneuver themselves in the environment, and to manipulate objects and their spatial locations. The OCEM methodology guides the agents to explore the environment using their actions and percepts, and facilitates building an environmental model incrementally until the agents can perform the given task. This methodology suggests a hierarchical architecture in which environmental models are learnt starting from an individual object to configurations of objects, and from shape level properties to task level properties. The model thus learnt is Object-centered, Grounded, Affordance-based, and Multi-layered (OGAM), and it is implemented as a multi-layered graphs. The evaluation of the learned model is based on the performance of "objects reconfiguration" task in the environment. In this task, the agents will use the learned model to recognize objects and familiar views of environment, to navigate themselves to the goal places, and to manipulate objects to construct simple configurations.

For self-organization issues, we have taken the hormone-based approach. We recognize that biological systems offer many valuable lessons for achieving the desired features of future agent systems, and the key enabling technology seems to be the ability of self-organization or self-reconfiguration. High-performance can be achieved by restructuring structures and better distributing tasks to resources. Robustness can be obtained by adapting to environments and self-repairing when single agents are damaged. Scalability can be accomplished by allowing autonomous agents to self-organize for global performance without pre-imposing any fixed superstructure. Furthermore, our earlier results in cell pattern formation have shown that homogeneous embryos cells can self-organize into complex patterns such as feathers and scales and the morphological differences between different patterns appears to be controlled not by the presence or absence of particular molecules but by the level and configuration of their expression. To demonstrate the generality of self-organization in robotics agents, we will use SOALA agents to autonomously reorganize in the process of learning and problem solving. During the past year, we have begun to formulate the *Digital Hormones Model* (DH-Model) for self-organization, and applied this approach to organizational problems in multi-agent systems such as action coordination and synchronization in self-reconfigurable robots. This report will give details on these issues.

During the period of 9/1/2001 through 8/31/2002, we have developed solutions for the two basic problems for the self-reconfigurable robots or systems: adaptive communication in self-reconfigurable and dynamic networks, and collaboration between the physically coupled modules to accomplish global effects such as locomotion and reconfiguration. Inspired by the biological concept of hormone, the Adaptive Communication (AC) protocol enables modules continuously to discover changes in their local topology, and the Adaptive Distributed Control (ADC) protocol allows modules to use hormone-like messages in collaborating their actions to accomplish locomotion and self-reconfiguration. These protocols are implemented and evaluated in the CONRO self-reconfigurable robot and a Newtonian simulation environment called 3D

Working Model. The results have shown that the protocols are robust and scaleable when configurations change dynamically and unexpectedly, and they can support online reconfiguration, module-level behavior shifting, and locomotion. These solutions can be generalized and applied to any distributed multiple robots and self-reconfigurable systems in general.

During the period of 9/1/2002 through 8/31/2003, we have developed solutions for two critical problems faced by self-reconfigurable systems: (1) Distributed Task Negotiation, that is, how agents negotiate and select a single critical task to execute when multiple tasks are proposed by different agents; (2) Distributed Behavior Selection, that is, how agents determine their local behaviors in order to produce a coherent global behavior. These solutions are parts of a new PhD dissertation by Behnam Salemi defended in June 2003 under the supervision of Dr. Wei-Min Shen. We describe the details of these solutions in the next section.

During the period of 9/1/2003 through 8/31/2004, we have developed solutions for two critical problems faced by self-reconfigurable systems: (1) Autonomous discovery and response to unexpected topology changes; (2) A new distributed functional language called DH2 for programming of self-reconfigurable systems using hormone-inspired computational methods. These results are published by Dr Wei-Min Shen and his students Behnam Salemi, Maks Krivokon, and Michael Rubenstein. We describe the details of these results in the next section.

3 Accomplishments and New Findings

3.1 2000-2001 Period

During the first year of the project (11/28/00 – 08/31/01), a substantial progress has been made towards the simulation of autonomous agents with percepts and actions suitable for a simulated 3D environment. We have selected the Award-wining Morfit (http://www.3dstate.com/) as the simulation environment and developed necessary programs to simulate agents. An interface has been developed and completed for agents to see and act in their environment. We have identified the necessary percepts and actions for the agents. Similar to having a camera and range detector, each agent can see an "image" of edges and distances to the edges. Each agent passes its action to the Morfit environment and our interface will carry out the necessary changes in the environment as the results of the execution of action. Each agent can perform four types of continuous actions: move-sideways, move-front-back, turn-body-around, and tilt-camera-vertically. As the foundation of the OCEM architecture, a graph structure and a learning algorithm have been developed and tested to build models for spatial and visible properties of object (we call them "object shape models"). In these models, each node is an internal representation of a percept and each link is an action. These models and algorithms are implemented in Microsoft Visual C++ 6.0 and tested on multi block environments. The programs are integrated with the 3D simulation environment, and the result is a synthetic agent that takes the visual data from the environment, builds object models in the environment, and uses the model to perform simple navigational tasks around the objects based on their shape properties. We elaborate each of the tasks in the following sections.

3.1.1 The Construction of Basic Autonomous Agents

This part of the project addresses the construction of basic autonomous agents that "live" in the 3D simulation environment. The Morfit application provides a 3D simulation of visual environment in which the agent learns and performs complex tasks. To implement the perception of agents, we have build a simulated camera program that takes the list of all polygons in the Morfit environment and converts them into a list of 2D edges that are "visible" in the camera image perceived by the agent. Each 2D edge is specified by a set of associated parameters - color, distance, spatial locations on the camera screen, and spatial points intersected by other edges.

Figure 1 illustrates the 3D environment and percepts received by an autonomous agent. The top-left window shows a top view of the entire environment in which there are three objects and one agent (the small black dot with yellow front). The top-right window is a rendered view (a bitmap) provided by the Morfit that is accessible to agent's camera. The lower-right window presents a list of edges that are received by the agent on its camera. These edges constitute the perceptual inputs to the agents. They are similar to the edges detectable by a vision program such as an edge-detection program. The bottom-left window is a control panel for directing agent's actions for interaction with the human users.

The Morfit requires additional functions in order to get the desired information from the environment and manipulate objects in the environment. Some significant function requirements are listed below.

- *Camera Function:* The percepts of an agent are the visible information provided by its camera, yet Morfit's functions return all the polygons available in the environment. We need additional functions that can extract the former from the latter each time the agent moves in the environment.
- *3D to 2D mapping:* The visual polygons make up a 3D world. The 2D edges received as percepts are the result of projection of the 3D polygons. Due to occlusion by the objects, not all edges, in front of the camera, are received as part of a percept. We need additional functions that map the 3D polygons to 2D edges, and find the visible parts of the 2D edges.
- *User Interaction:* In order for the user to interact with the interface to test the agent performance, we need a GUI to specify and control agents' location and movement. For example, the user should be able to say that the agent start at a particular location and move in a direction for some steps. The visual images seen by the robot's camera will be updated each time a movement is executed.

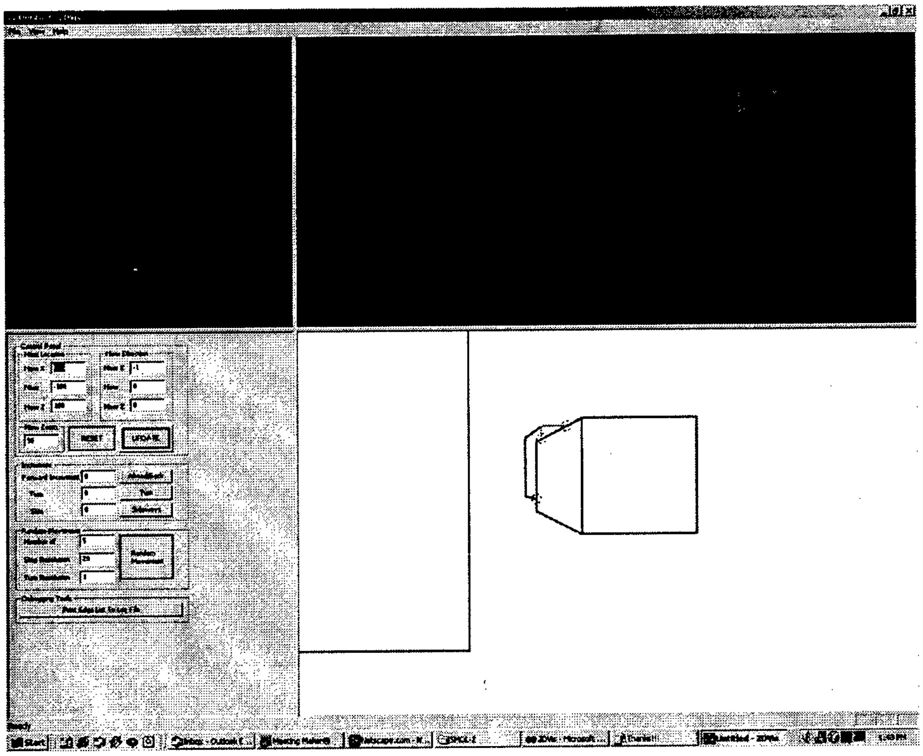


Figure 1: A view of Morfit 3D world and a perceived image of edges by the agent

In order to separate the polygons into those that are in front of the camera and those that are behind the camera, we need an efficient algorithm. We have implemented a Binary Search Partition (BSP) tree generation and BSP tree traversal algorithms. The BSP tree organizes the polygons in the world as a tree where the relationship between two successive nodes is either "in front of" or "behind". Using these relationships, the BSP traversal algorithm can find the list of polygons that are in front of a camera.

The list of the 3D polygons, generated by BSP tree search, is then converted into 2D edges. We implemented a graph algorithm that detects the hidden edges and eliminates them. The input to this algorithm is a list of 3D points that are in front of the camera. The output is a list of 2D points that are imprinted on the screen of a camer

In order to allow the user to interact with the agent and the environment, we have designed a GUI. Figure 1 also illustrates a screen shot of the GUI (bottom left corner). The user can select a starting location for the agent, and can direct the agent's action by running a complete simulation, or proceeding step by step. The user can also ask the agent to behave as a reflexive agent, by selecting actions arbitrarily, or allow a learning algorithm to guide it.

Shape Model Building for Single Objects

The autonomous agent learns to perform a complex task in a new environment by building an OGAM model for predicting the results of its actions. The model is built incrementally in abstraction. In this first phase of the project, we developed an algorithm that builds the shape model of single objects from the percepts and the actions. The shape model of an object allows the agent to perform several tasks like navigation around the object, prediction about the consequences of its navigational actions, and the recognition of the object.

There are a number of issues in building the object shape model in a continuous environment:

- ***Infinite Exploration Space***: In order to build a model of the object that can help the agent navigate from any point to any other point in the environment, the agent may need to explore infinite points. This is because no two points provide same relationship information between the agent and the object. This is especially difficult for an autonomous agent does not have a model that can relate two points, such as a global coordinate system.
- ***Infinite action space***: The space of actions from which the agent can select is infinite. The agent we have designed can perform four kinds of continuous actions: {move-sideways, move-forward-backward, turn-horizontally, tilt-camera-vertically}, and emulate all possible parameters for these actions will be impossible.
- ***Large Relationship Space***: The number of relationships among the objects and an agent is very large. For example, the distances between the agent and an object can vary widely, so are the perspectives between the agent and the object.

Our methodology has two major steps in building the shape model for navigation about the object. The first step is to build a "core" model within a restricted relationship space. That is, the distance is kept within a limit while changing the rest of the relationships. The objective of this core model building is for the agent to acquire the maximal information of the edge features of the object, with a minimal variation of relationship. The second step is to generalize the core model over the entire space of relationships. In this report, we describe the first step only.

The agent needs to navigate in the relationship space with the object. The relationship space of an agent with respect to an object is defined by the edges of the object that are visible to the agent, their perspectives, and the distance between the agent and the edges. Our approach is based on a significant insight that a complete shape model of an object for the entire relationship space can be generated from a more specific model that covers the complete feature space of the object but within a narrow relationship space, by generalizing over distance and perspective relationships. Core model is such a specific model where the distance relationship between the agent and the object is kept within a limit while spanning the feature space rapidly.

Our algorithm for core model building has the following novel features.

- *Affordance-based representations:* The agent should be able to select actions based on the most recent percept it has received. The action selection should be based on a task such as building a core model for a restricted relationship space. In order to link the percepts, with 2D edges, and the actions, which are expressed as number of pulses sent to the effectors of the agent, the percepts are converted into a parametric form where the action parameters are used to represent the 2D edges. We use *prototypes* that generate edges as functions relating the parameters of the actions. These prototypes are themselves represented in a form that can be learnt empirically by the agent.
- *Prototypes for generating grounded representations:* For a model state, the agent should be able to assess the extent of the physical or simulated world covered. This mapping enables the agent to synchronize its traversal within its model space with the physical

movement. It also enables the agent to match the physical data with parts of its model states. In order to address these issues, the agent starts with a set of prototypes one for each type of action. These prototypes represent each action parameter (such as move sideways) as a function of the distance between the agent and the real feature in the environment. The distance is measured in terms of the action parameter units. For example, move_sideways = 2*d*cos(90-(ϕ /2)), where ϕ is the angle of the camera, d is the distance measured in terms of pulses given to the agent's motor. It can be easily shown how the distance can be mapped onto the action parameters. Each edge feature in model stat is generated by instantiating a prototype. Thus an edge feature can be used to find the effects of executing an action as changes in that edge feature.

- *Measuring the maximal change in feature space (E_{cm}):* The purpose of the core model building is to achieve maximal change in the feature space by minimal actions. We use a parameter, Model Change Estimate E_{cm} to calculate the maximal change possible by each action. E_{cm} is a sum of several changes possible in the current model state the agent has due to the execution of an action. E_{cm} is given as the sum of (1) the changes in the lengths of the edges, (2) the positional changes of the edges, (3) the orientation changes of the edges, and (4) the amount of disappearance of an edge from view field. The agent selects the action that has the largest E_{cm} value. The significance of this step is that it maps large number actions possible at each step to a small set of discrete actions.
- *Continuity Condition*: In core model building, the agent is not trying to build a geometric model of the object. Instead, it is building a model that allows predictions about the properties of objects as a consequence of executing its navigational actions. Any model built by an agent should allow some measure of progression towards o away from a goal state. A model constructed by a randomly moving agent cannot provide this measurement. We use an insight that the models of the object features allow such a measurement. In order to build this measurement space, the agent needs to scan the object by maintaining continuity over the edge features. This method is far superior to the one that maintains an abstract parameter for measurement as the validity of the continuity of this parameter is not known. The continuity condition is incorporated into E_{cm} calculation. That is, the agent will not consider an action as a potential candidate for execution if it does not allow the point features from the current model state to be incorporated into the next state.
- *Perspective Changes:* As the agent explores for the data in a 3D world, it needs to change its tactics to get useful data, especially to satisfy the continuity condition. This can happen when it reaches the corner of a block. Then the agent has to move in order to access the connecting surface features. The agent needs to recognize such situations and then act accordingly. Our approach is to recognize such situations by significant changes in the perspectives of the edges in the current model state. Our algorithm calculates such changes and then modifies E_{cm} to select actions that would allow it to explore a new face of the object.
- *Terminating the core model building:* The agent needs to address the question of how to terminate the model building. Since the agent uses the grounded and Affordance-based representations, this is not a serious problem. The agent maintains the cumulative action parameter values in its model history. These parameters exhibit classical geometric properties. For example, turning right will eventually make the agent end up in the same

direction. The agent imposes a set of terminating conditions on the history of model states it has been developing.

Figure 2 illustrates the overall algorithm for the core model building.

```
0.0     $p_{in}$ ← initial_percept
1.0     $m_{in}$ ← generate_model_state ($p_{in}$, prototypes, NULL)
2.0     history ← add_to_history($m_{in}$)
3.0     action ← select_action($m_{in}$, action_types, prototypes)
4.0     percept ← execute_action(action)
5.0     model_state ← generate_model_state(percept, prototypes, history)
6.0     history ← add_to_history(model_state)
7.0     condition ← estimate_termination_condition(history)
    7.1     if (condition == true)
            7.1.1   Exit
    7.2     Else, go to step 3.0
```

Figure 2. The overall algorithm for core model building

Integration

We have integrated the core model building with the graphics interface so that the agent gets a list of 2D edges as percepts from the Morfit environment and can pass an action to be executed in the Morfit environment. Figure 3 illustrates how this integration is made.

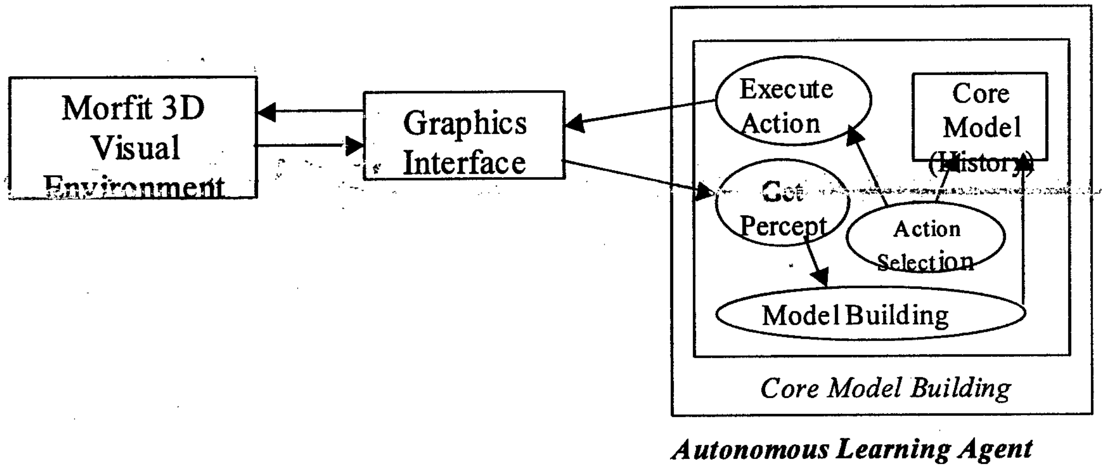

Morfit 3D Visual Environment
Graphics Interface
Execute Action
Core Model (History)
Get Percept
Action Selection
Model Building
Core Model Building
Autonomous Learning Agent

Figure 3. Integrating Core model building with the graphics interface

Evaluation of Object Shape Model

To evaluate the performance of the autonomous agents, we plan to give the agent certain recognition and navigation tasks once it has learned the core model. One scenario is to present a image of an object to the agent and the agent is required to navigate to a location in the environment where this image can be received by the camera. To accomplish this task, the learning agent must use the learned shape model to guide its navigation or further exploration of the environment. The agent must first compare the goal image to all the model states that have been learned, and determine if this goal image is contained in the core model (i.e., a place that has been visited before). If so, the agent has to plan a sequence of actions to navigate itself to the goal place. Otherwise, the agent must find the most similar place in the model, navigate there, and perform more exploration in search for the goal image. To accomplish such a task, the agent should first be able to generalize between most similar state and the goal image. Since the generalized model has not yet been evaluated by the agent, the model can only suggest a navigational path to the goal image place with certain amount of uncertainty. Further, the model states that may be encountered by the agent on its way to the goal image pace may not be exactly same as the once suggested by the generalized model, the generalized model states can match with those encountered with some amount uncertainty. We have been trying to incorporate some probability measures to address these two problems.

3.1.2 The Digital Hormone Model (DH-Model) for Self-Organization

Based on our previous research in cell development and metamorphic robots, we developed a biologically inspired computational model for self-organization. In this model, pattern formation of multiple elements is based on the behavior rules of single elements and mediated through hormone-like signals (digital hormones). It is a **bottom-up** patterning model because various configurations can be produced based on the behavioral response of individual elements to the local information in the environment. This is in contrast to the **top-down** approaches where generation of pattern is considered as a readout of a global coordinate system, or key-lock like chemo-affinity (Meyer 1998). The top-down models indeed work in some biological situation. In case such as Drosophila segmentation formation, it was found to be based on zip code like specific DNA sequence in the enhancer regions (Struhl 1992). However, these "blueprint" models have an obvious problem in terms of information load. As animals evolve and become more and more complex, the enormous information required will soon over-burden the system. There must be other ways in biology that are used to build complexity and diversity. The fact that the size and spacing of leopard spots and zebra stripes are consistent (e.g., similar size, spacing) but non-identical implies that the pattering process follow rules, not a blueprint. Our model is unique in that ***it is based on rules, not coordinates.*** As a result, much less information is required, and it is flexible and can accommodate environmental changes and re-configurate quickly. This is because each element can make its own response to the ever-changing local information, not have to search for preset instructions. These rules are part of the "bio-informatics" we need to learn from Nature.

The Digital Hormone Model. Our self-organization model (see Figure 4 and Table 2) takes place in a **space** of grids. There are two basic types of elements: a set of cells and digital hormones. There is a global ***clock*** that set the patterning process in motion. The space can be of fixed size or expandable. A certain number of ***cells*** is added to the system initially, but cells numbers may increase or decrease during the process. We assume that each cell occupies one grid, and each grid can accommodate only one cell but multiple digital hormones. Each cell is an autonomous system that has certain properties. One major property is that it can secret digital hormones and has receptors to digital hormones. ***Digital hormones*** are typed elements that can be released from cells into the grids or captured by receptors of cells from the grids. Each type of digital hormone has its own density threshold and diffusion function. Digital hormones are propagated in the space from the higher density grids to the lower density grids with the ratio specified by the diffusion function. The propagation stops when there is no gird with density higher than the threshold. Within a grid, the number of typed digital hormones that will be bound to the receptors of the occupying cell equals the number of the same type receptors in the cell. The receptors have different receptor types and can bind digital hormones with matching types. Receptors can be "used up" when bound to digital hormones, and can be created or deleted by cell's actions. Thus the number and types of receptors in a cell may vary in a life time.

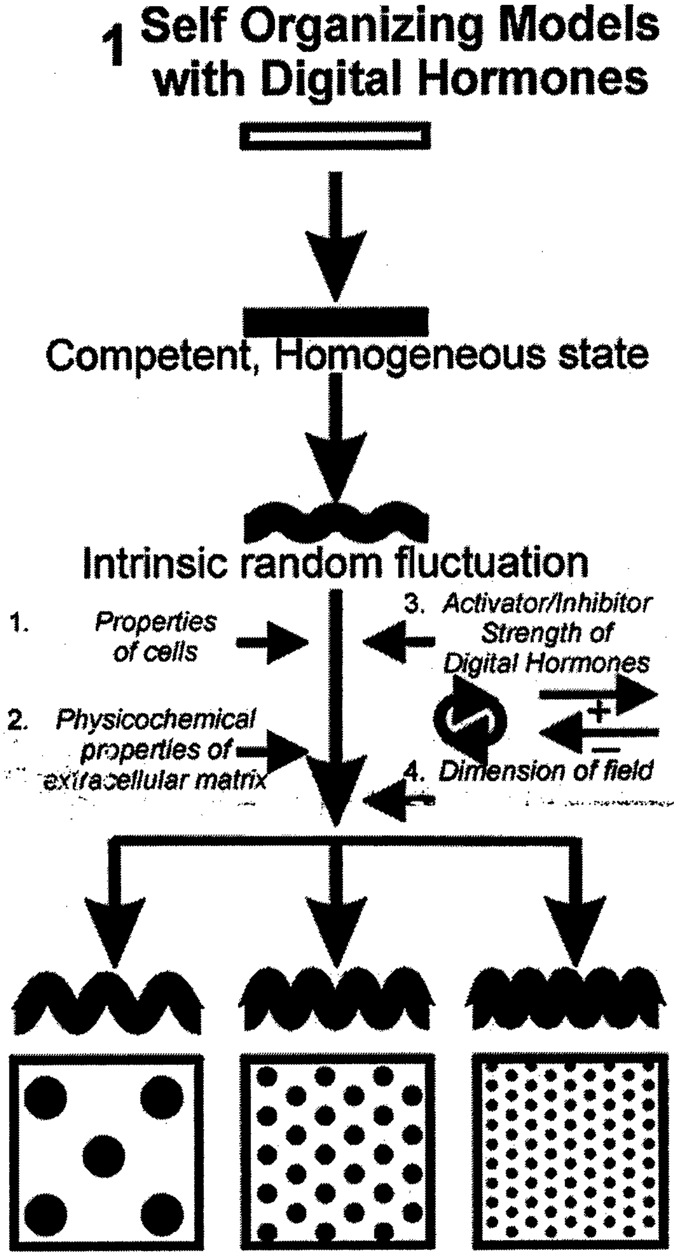

1
Self Organizing Models with Digital Hormones
Competent, Homogeneous state
Intrinsic random fluctuation
1. Properties of cells
2. Physicochemical properties of extracellular matrix
3. Activator/Inhibitor Strength of Digital Hormones
+
−
4. Dimension of field

Figure 4: Self-Organizing Models with Digital Hormones

The variables of a cell reflect the state of cell property, which include adhesion information, migration, differentiation (converted to different types of cells), and so on. An ***adhesion*** action causes the cell to be connected or disconnected from a neighboring cell. The adhesion level between two neighboring cells can be attractive (with positive adhesion value, further defined in Figure 5), repulsive (with negative adhesion value) or neutral (zero value). Digital hormones can alter the strength of connection, and are classified as activators or inhibitors depending on its ability to increase or decrease cell connection. A ***migration*** action causes the acting cell to move one grid in a specified direction. The migration can be random or directed, and a group of connected cells can move synchronously. If the target grid is already occupied, this action has no effects. A ***secretion*** action causes the cell to release certain number and types of digital hormones. A ***modification*** action changes the number and types of receptors in the cell. A ***proliferation*** action creates new cells, which may be naive cells or copies of certain acting cells. A ***differentiation*** action makes the cell change values for the specified variables such as shape or size. A ***death*** action terminates the life of the cell and remove it from the occupying grid.

Table 2: Digital Hormones for Biological Pattern Formation and Robot Reconfiguration

DH-Model for Self-Organization	**Feather pattern formation**	**Metamorphic robots**
SPACE: Expandable evenly or unevenly vs. Fixed	Skin on embryonic chicken	Terrain
TIME: A global clock	Time in development	Distributed clocks
CELLS	Biological cells	Robotic cells
# of Cells: Increase or decrease	Proliferation, Cell death	Addition, Destruction
Cell Connection Index (Ci): 0: neutral; 2.5: weak connectivity 5: can connect; 7.5: connected reversibly 10: connected irreversibly Negative value: repulsive, defined similarly	Cell adhesion e.g., NCAM cell adhesion	Connection between robot modules
Cell Migration Random or directed, Aggregate or cluster migration	Cell migration Cell group migration	Robot movement Configured robot movement
Digital Hormones Activator (A) for connection Inhibitor (I) for connection Modifier (M)	Secreted extracellular signaling Molecules, e.g., A: FGF; I: BMP; M: sprouty, noggin	Infrared Radiowave
Cell State Activator receptor (AR) Inhibitor receptor (IR) Productivity of A and I Differentiation (converted to other cell types)	Number of receptors on cells Rate of secretion	Sensitivity to receive signals, Strength of emitted singals
RULES of digital hormone behavior Spread out radially from cells Strength decreases with distance A and I travels with different velocity	Turing model Diffusion rate Property of extracellular matrix	To be applied
POSSIBLE CONFIGURATIONS Aggregates, Cluster, Stripes, and others	Feather patterns, leopard spots, Zebra stripes	Tetrapod, snake, Sea urchin, etc.

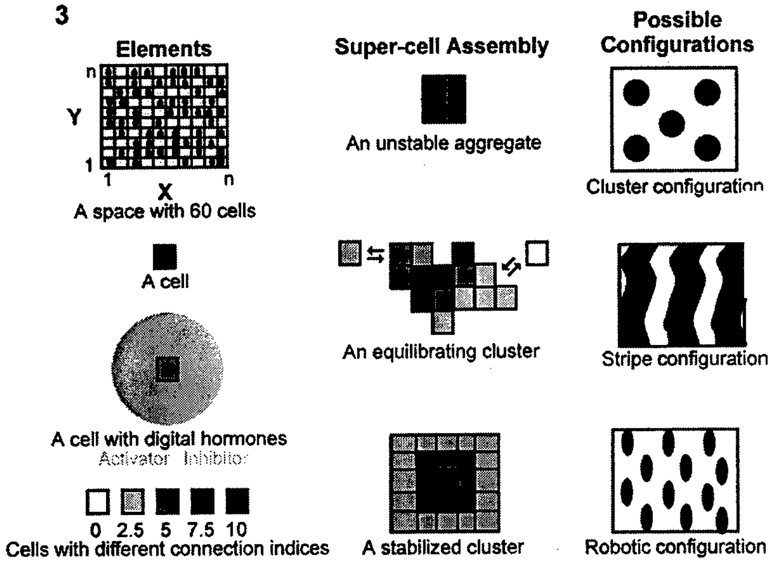

3
Elements
Super-cell Assembly
Possible Configurations
n
Y
1
1
X
n
A space with 60 cells
An unstable aggregate
Cluster configuration
A cell
An equilibrating cluster
Stripe configuration
A cell with digital hormones
Activator Inhibitor
0 2.5 5 7.5 10
Cells with different connection indices
A stabilized cluster
Robotic configuration

Figure 5: An example of self-organization with digital hormones

The entire computational model works as follows. Once the clock starts, the whole model will perform the following events repeatedly:

1. All cells will bound their receptors to the existing digital hormones in their grids;
2. All cells will select actions according to their behavior rules;
3. All cells will execute their actions;
4. All digital hormones in the space will be propagated by their diffusion functions.

The selection and triggering of these actions are governed by the ***behavior rules*** that map the conditions of the cell to the actions. For example, one rule might say "if there are more than *n* bound receptors of type *x*, then increase the receptors of type *y* by *m*." To illustrate how this model can be used to experiment self-organization, consider a simple example of pattern formation where cells are checker pieces of black, gray, or white, and the space of grids is the checker board (Fig.5; Empty space in light yellow). Suppose that the pieces can change their colors in accordance to its adhesive state (Fig.5 left lower panel) and can self-organize into certain configurations (Fig.5, right column) from a homogeneous initial state. We will set up a computer simulation model, and test and modify this model. Our challenge is to design cells, digital hormones and their receptors, and to identify and set behavior rules for the cells so that such self-organization can be accomplished. We can experiment by varying these parameters and observe their effects on the self-organization process. For example, according to Turing model (Turing 1952), a random fluctuation in a homogeneous system can be stochastically amplified and lead to periodic pattern formation, with dots or stripes, of different size and spacing. In our computer simulation, we can set a naive state. For example, we can assign connection index 5 to all cells, and the same production rate of digital hormones. We can then allow the state of cell connection fluctuate with $\pm$ 1, or even $\pm$ 0.1 range, then set the clock and see what stable configuration it will develop (Fig.5, middle column). ***Stable configuration*** is defined as the maintenance of a configuration and is the result of competition and equilibrium. The computer simulation will also allow us change other variables such as level of digital hormones, diffusion rate of digital hormones, ratio of activators versus inhibitors, number of receptors on cells, changes of connection index by digital hormones, etc. one at a time. We can analyze the change of configurations, cluster versus size, the size of clusters and spacing, radially symmetry versus

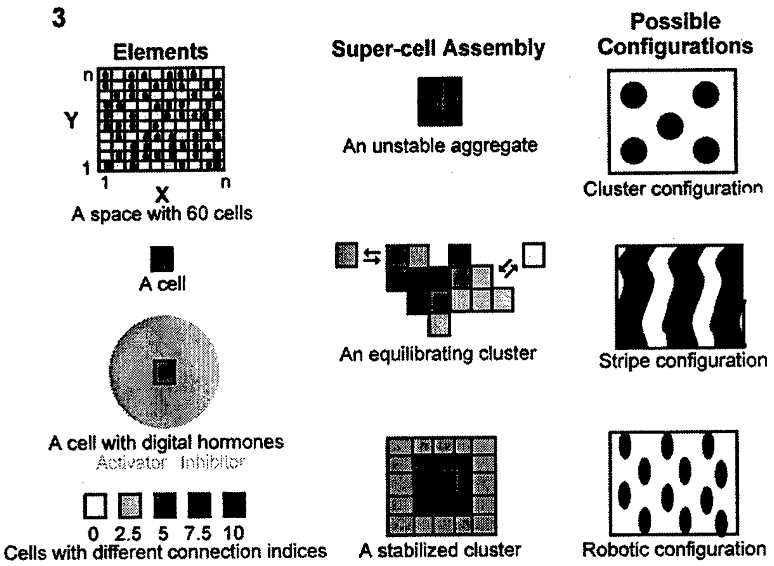

3
Elements
Super-cell Assembly
Possible Configurations
n
Y
1
1
X
n
A space with 60 cells
A cell
A cell with digital hormones
Activator Inhibitor
0 2.5 5 7.5 10
Cells with different connection indices
An unstable aggregate
An equilibrating cluster
A stabilized cluster
Cluster configuration
Stripe configuration
Robotic configuration

Figure 5: An example of self-organization with digital hormones

The entire computational model works as follows. Once the clock starts, the whole model will perform the following events repeatedly:

1. All cells will bound their receptors to the existing digital hormones in their grids;
2. All cells will select actions according to their behavior rules;
3. All cells will execute their actions;
4. All digital hormones in the space will be propagated by their diffusion functions.

The selection and triggering of these actions are governed by the ***behavior rules*** that map the conditions of the cell to the actions. For example, one rule might say "if there are more than *n* bound receptors of type *x*, then increase the receptors of type *y* by *m*." To illustrate how this model can be used to experiment self-organization, consider a simple example of pattern formation where cells are checker pieces of black, gray, or white, and the space of grids is the checker board (Fig.5; Empty space in light yellow). Suppose that the pieces can change their colors in accordance to its adhesive state (Fig.5 left lower panel) and can self-organize into certain configurations (Fig.5, right column) from a homogeneous initial state. We will set up a computer simulation model, and test and modify this model. Our challenge is to design cells, digital hormones and their receptors, and to identify and set behavior rules for the cells so that such self-organization can be accomplished. We can experiment by varying these parameters and observe their effects on the self-organization process. For example, according to Turing model (Turing 1952), a random fluctuation in a homogeneous system can be stochastically amplified and lead to periodic pattern formation, with dots or stripes, of different size and spacing. In our computer simulation, we can set a naive state. For example, we can assign connection index 5 to all cells, and the same production rate of digital hormones. We can then allow the state of cell connection fluctuate with ± 1, or even ± 0.1 range, then set the clock and see what stable configuration it will develop (Fig.5, middle column). ***Stable configuration*** is defined as the maintenance of a configuration and is the result of competition and equilibrium. The computer simulation will also allow us change other variables such as level of digital hormones, diffusion rate of digital hormones, ratio of activators versus inhibitors, number of receptors on cells, changes of connection index by digital hormones, etc. one at a time. We can analyze the change of configurations, cluster versus size, the size of clusters and spacing, radially symmetry versus anterior-posterior asymmetry, etc. New elements and rules will be added or modified as we experiment. In this grant, we will first focus on the interplay among cell adhesion, migration and digital hormones, and leave other parameters as constants. After we can master the prototype model, we could modulate others variable to build or simulate more complex patterns and phenomena. This include making "space" expand evenly or unevenly, adding new cells randomly in space or from a localized input, or defining differentiation so that multiple types of cells co-exist and interact (such as the differentiation of stem cells into tissues, or evolution of single cellular organism into multi-cellular organisms).

The Unique Features of the Digital Hormone Model. The digital hormone model proposed here has many advantages over other models for self-organization and self-reconfiguration because it is distributed, scalable, robust, adaptive, and easy to develop. The model ***does not require a fixed control center***. In fact, any cell can play the leader role for a particular cooperation. This advantage comes from the fact that no individual cell is required to have addresses or identifiers, and no single special cells are required for pattern formation. Communication is established by the releases and receptions of digital hormones, and the receiver cells can autonomously decide their actions based on their local information and rules. No individual needs to know the global configuration of the system. The model is ***scalable*** because the functions of a cell in the configuration (or in a organization) can be modified in a completely autonomous manner, regardless of the existence and states of other cells in the organization. This enables every cell of the organization to have autonomy in sending and receiving messages, and accepts changes of organization in a very flexible manner. Since the behaviors of an individual cell are determined only by its local information (such as how many neighbors it has and what type of neighbors they are) and its internal state and knowledge, a part of an organization can change without affecting the rest of the organization. Similarly, in order to add new cells into an existing configuration, all it requires is to place them into the living space (such as done by the proliferation action). No individual cell is required to know who or how many existing members are and what these members can do. Such will be a highly desirable feature for the ever expanding information systems. The model is ***robust*** for self-reconfigurable cooperative organizations. Since digital hormones do not require any centralized control, malfunctions of individual cell will not cause collapse of the entire organization and local failures will not have catastrophe effects on global performance. Furthermore, if the organization is capable of reconfiguration, then damaged cells can be detected and discarded without affecting the performance of the whole organization. The model can support organizational ***adaptation.*** This is because the linkages between members are flexible, changeable, and controllable; they can dynamically form new configurations to meet the demands of applications or changes of the environment. Finally, this model suggests a hormone-oriented and cell-oriented programming paradigm for distributed ***software development***. Since cell-based programs are modularized and can be developed in parallel, they may be easier to develop, more reliable, predictable, modifiable, and adaptable than most conventional programming paradigms such as the object-oriented programming.

The DH-Model has many similarities to but is different from the traditional cellular automata model. They both use cells, grids, and rules, but the DH-Model has complex hormone propagations and cells can connect to each other and move as aggregates. Given the evidence that John Conway's two simple rules for the Life game can produce complex behaviors, the DH-Model is indeed capable of simulating sophisticated behaviors.

The DH-Model is also different from amorphous computing (Abelson 1999; Nagpal 1999), which is a system of irregularly placed asynchronous, locally interacting computing elements coordinated by diffusion-like messages and behave by rules and state markers. However, amorphous computing is not a study of self-organization but an engineered system for elements to organize and behave in a priori intend. Furthermore, their coordinate system assumes that positional information is the key for pattern formation (Wolpert 1969), while our DH-Model emphasizes configuration information.

3.1.3 Self-Reconfigurable Robots and CONRO Modules

Among all computational studies of self-organization, modular self-reconfigurable robots are perhaps the only one that implements and demonstrates self-organization on physical devices. These robots are constructed from a set of autonomous robot cellular modules that can self-reconfigure into structures of different shape, size, and configuration in order to accomplish complex task in dynamic and uncertain environments. These robots are highly desirable for tasks such as fire fighting, search/rescue after earthquake, and battlefield reconnaissance, where robots would encounter unexpected situations that are difficult for fixed-shape robots to deal with. For example, to maneuver through difficult terrain, a metamorphic robot may have to transform itself into a snake to pass through a narrow passage, grow legs to climb over obstacles, or become a ball to roll down a slope. Similarly, to enter a room through a closed door, a metamorphic robot may disassemble itself into a set of smaller units, crawl under the door, and then reassemble itself in the room.

Research in self-reconfigurable robot started in 80's when Fukuda and Kawauchi (Fukuda 1990) proposed a cellular robotic system to coordinate a set of specialized modules with a distributed control method. Yim (Yim 1994) has designed a set of mechanical modules for self-reconfiguration and proposed the mechanism of gait control tables and open-loop synchronization. Chen (Chen 1994) has studied the theory and applications of modular reconfigurable robots and discussed the concepts of connecting mechanisms. Murata et al. (Murata 1994) and Yoshida et al. (Yoshida 1997; Yoshida 1998) designed and implemented several distributed control methods for 2D and 3D self-assembly and locomotion. Paredis and Khosla (Paredis 1995) proposed modular components for building fault-tolerant multipurpose robots. Neville and Sanderson (Neville 1996) proposed a module for the dynamic construction of complex structures. (Chirikjian 1996; Pamecha 1997) has studied the metric properties of reconfigurable robots. (Kotay 1998) designed and built a set of robot molecules. Lee and Sanderson (Lee 1999) have proposed a distributed control method for Terobot modular robots. Fujita, et. al. (Fujita 1998) describe a robot dog system that can be manually assembled into different configurations. Bojinov et al. (Bojinov 2000) proposes a multi-agent approach to control metamorphic robots for object manipulation and reaction to external forces, but the "scent" concept used in their system has only limited capabilities for module cooperation and requires global synchronous updates. Kotay and Rus (Kotay 2000) have proposed a set of rule-based finite state automata. Recently, two workshops (Rus 2000; Shen 2001) and a special issue of Autonomous Robots journal has devoted to this exciting topic.

In the past three years, we have built a new generation of self-reconfigurable robots called CONRO. The CONRO robot consists of a set of modular modules that can connect/disconnect to each other to form different complex structures for different tasks. Each CONRO module has a size of 1.0 inch^2 cross-section and 4.0 inch long and is equipped with a micro-controller, two servo motors, two batteries, four connectors for joining with other modules, and four pairs of infrared emitter/sensor for communication and docking guidance. In comparison of other existing metamorphic robots, the unique properties of CONRO robot modules are that they are autonomous and completely self-sufficient with own power, computational resources, and sensory and actuating devices, and they have the automatic docking capability to connect and disconnect with each other to form various shapes and size.

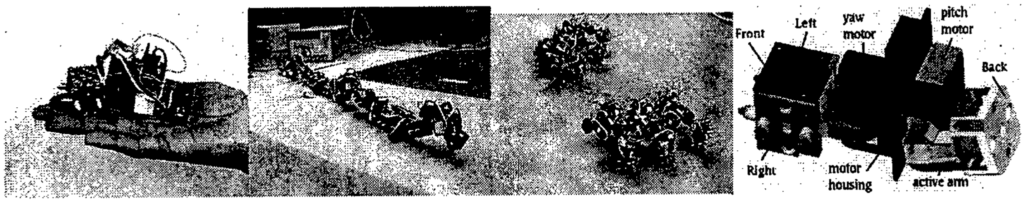


Figure 6: Examples of CONRO Self-Reconfigurable Robots

The four pictures in Figure 6 show a CONRO module in hand, an 8-module snake, two 9-module six-legged insects, and the detailed schema of a single CONRO module, respectively. For more information and movies of CONRO, please visit the web site http://www.isi.edu/conro and see (Castano 2000; Castano 2000; Shen 2000; Shen 2000; Shen 2000; Salemi 2001; Shen 2001). At the present time, 20 CONRO modules have been built. These modules can be connected to form various configurations including snake, caterpillar, quadruped, and hexapod. These configurations are capable of performing basic locomotion and self-reconfigurations.

Shown in the detailed schema of CONRO module, the four connectors are located at either end of each module. At one end, called the module's *back*, is a female connector, consisting of two holes for accepting docking pins of other modules. At the other end, three male connectors, each has two docking pins, are located on three sides of the module, called *front, left,* and *right.* The female connector has an SMA-triggered locking/releasing mechanism. Each module has two degrees of freedom: pitch and yaw. When two or more modules connect to form a structure, they can accomplish many different types of locomotion. For example, a body of six legs can perform hexapod gaits, while a chain of modules can mimic a snake or a caterpillar motion. CONRO modules communicate with one another using IR transmitters and receivers located at the connectors. When a module is connected to or near another module via a connector, the two pairs of IR transmitters/receivers at the corresponding connectors will be aligned to form a bi-directional communication link. Since each module has four connectors, each module can have up to four communication links.

Although metamorphic robots have a wide range of applications, the control of such robots, however, is not a trivial task. The difficulties stem from the facts that all locomotion, perception, and decision making must be distributed among a network of modules, that this network has a dynamic topology, that each individual module has only limited resources, and that the coordination between modules is highly complex and diverse. For example, communication is difficult because such robots can autonomously and possible frequently change the connections among components. These changes alter the communication channels between modules and modify the topology of the underlying communication network. Another difficult problem is cooperation. Individual modules can only access their neighbors' local information yet all modules must function according to their roles and locations in the current configuration. We cannot assume any single module to be the fixed "brain" of the system because damages to the brain module will paralyze the entire population. Furthermore, actions of individual modules are typically weak in comparison with the entire robot body, and no single module can be assumed to be powerful enough to manipulate all other modules in the system. Thus the weak and local actions must be coordinated to produce strong and precise global effects. Finally, without a global clock for all modules, how actions in many different modules can be synchronized is an interesting and challenging question.

3.1.4 Applications of DH-Model to Metamorphic Robots

The DH-Model can be readily applied to the control of metamorphic robots. Consider a set of metamorphic robots that are made of robotic cells (r-cells) that can connect and disconnect with each other to form different configurations or organizations. We assume such r-cells have the similar actions as those proposed in the DH-Model but live in a free space without grids. All r-cells have the same internal structure as shown in Figure 7, where software and hardware are constructed to simulate the biological receptors and the relevant part of decision-makings process. A local engine with a set of receptors can examine the incoming signals received from its active links, and decides if any local actions should be taken. Such actions include activating local sensors and actuators, modifying local receptors or programs, generating new digital hormones, or terminating digital hormones. Just as a biological cell, a r-cell's decisions and actions depend only on the received hormones, its receptors, and its local information and knowledge.

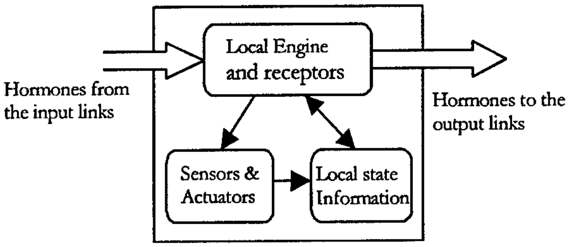

Local Engine and receptors
Hormones from the input links
Hormones to the output links
Sensors & Actuators
Local state Information

Figure 7: Internal structure of a r-cell

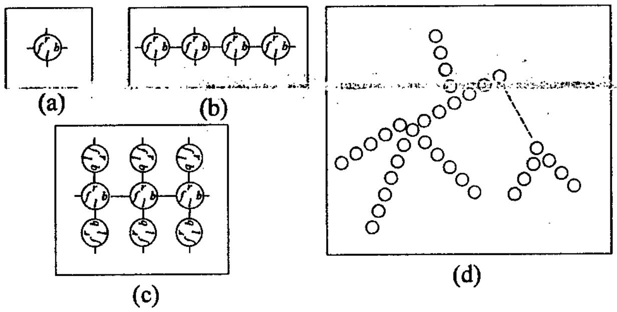


Figure 8: Examples of r-cell Organization

We represent a configuration of r-cells as a graph with nodes as r-cells and edges as established connections. For example, a single CONRO-like r-cell with four potential connectors can be represented as the graph shown Figure 8(a) where all four connectors are open. A graph for a snake-like chain of four r-cells can be represented as a graph in Figure 8(b), a 6-legged insect in Figure 8(c), and a system with two separate robots with a remote communication link (dashed line) in Figure 8(d). In general, an organization of r-cells can be an arbitrary graph with r-cells having different number of connections.

The digital hormones can be used to accomplish the communication, collaboration, and synchronization among r-cells. For simplicity, we only consider a special case of DH-Model where hormones are mainly travel in the connections between r-cells and the diffusions through free space are negligible. From a computational point of view, a digital hormone in this case is a message propagating in the r-cell network and it has three important properties: (1) a hormone has no destination; (2) a hormone has a lifetime; and (3) a hormone contains codes that can trigger different actions at different receiving r-cells.

To illustrate the application of DH-Model in metamorphic robots, consider an example how digital hormones are used in self-reconfiguration. Figure 9 illustrates a situation where a metamorphic robot with seven r-cells changes from a quadruped (a four legged structure) to a snake. In this figure, a r-cell is represented as a line segment with two ends: a diamond-shaped end (the back link) and a circle-shaped end (this end has three possible links: the front, left and right). The robot must change from a legged configuration (at the top-left of the figure) into a snake (at the bottom of the figure). To do so, this robot must perform the "leg-tail assimilating" action four times. To assimilate a leg into the tail, the robot first connects its tail to the foot of a leg and then disconnects the leg from the body (shown at the upper part of the figure). Just as in any r-cell organization, each r-cell in the robot determines its role based on its local state information including its own neighboring connections.

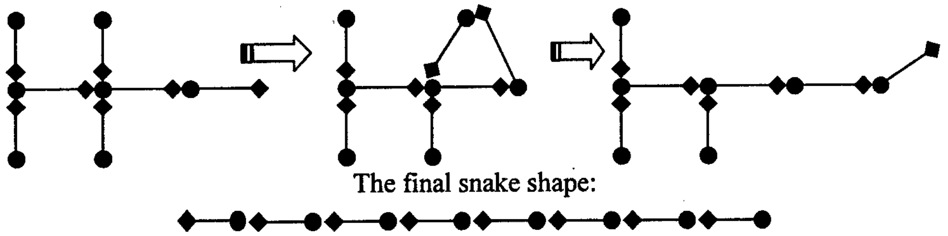


Figure 9: Reconfiguration from Quadruped to Snake

Digital hormones	**Actions**
LTS	Start the reconfiguration
RCT_1, RCT_2, RCT_3, RCT_4	Legs are activated
TAR, RCT_2, RCT_3, RCT_4	The tail inhabits RCT, and leg1 determines RCT_1
ALT, RCT_2, RCT_3, RCT_4	The tail assimilates leg1 and then accepts new RCT
TAR, RCT_2, RCT_4	The tail inhabits RCT, and leg3 determines RCT_3
ALT, RCT_2, RCT_4	The tail assimilates leg3 and then accepts new RCT
TAR, RCT_2	The tail inhabits RCT, and leg4 determines RCT_4
ALT, RCT_2	The tail assimilates leg4 and then accepts new RCT
TAR	The tail inhabits RCT, and leg2 determines RCT_2
ALT	The tail assimilates leg2 and then accepts new RCT
$\varnothing$	End the reconfiguration

Using digital hormones, the entire reconfiguration procedure starts when one (and any one) of the r-cells generates a reconfiguration digital hormone LTS (Legs To Snake). This LTS digital hormone is floating to all r-cells, but each r-cell's reaction to this LTS digital hormone will be different and that depends on the receiver's role in the current configuration. For this particular digital hormone, no r-cell will react except the foot r-cells, which will be triggered to generate a new digital hormone RCT (Requesting to Connect to Tail). Since there are four legs at this point, four RCT digital hormones will be floating in the system. Each RCT carries a unique signature for its sender. No r-cell will react to a RCT digital hormone except the tail r-cell. Seeing a RCT digital hormone, the tail model will do two things: acknowledge the RCT by sending out a new TAR (Tail Accept Request) digital hormone with the signature received in the RCT, and inhibit its receptor for accepting any other RCTs. The new TAR digital hormone will reach all r-cells, but only the leg r-cell that initiated the acknowledged RCT will react. It first terminates its generation of RCT, and then generates a new digital hormone ALT (Assimilating a Leg to the Tail) and starts the required reconfiguration action (see (Shen 2000) (Shen 2001) for the details of this compound action). When seeing an ALT digital hormone, the tail r-cell will terminate the TAR digital hormone and starts actions to assimilate the leg. After the action is done, the tail r-cell will reactivate its receptor for RCT digital hormones, and another leg assimilation will be performed. This procedure will be repeated until all legs are assimilated.

As we can see from this example, applying DH-Model to metamorphic robot control results in a number of advantages. First, the method works in many different configurations. In our current example, it will work independent of the number of legs in the system and how long the tail is. Second, the digital hormones are naturally organized in hierarchical structures. For example, a single LTS can trigger a level of activity managed by the digital hormones RCT, TAR, and ALT. One ALT will trigger another level of activity for assimilating a leg using another set of digital hormones (we did not show the details of this level in this example). Third, digital hormones allow global actions to be totally distributed to individual members. All r-cells have total autonomy in deciding their local actions, generating or terminating hormones. This allows on-line reconfiguration where a robot can maintain its function when merging or disconnecting with other robots. Fourth, this approach is de-centralized and any r-cell in the configuration can serve as the trigger of the reconfiguration. it is more efficient than centralized approaches because one single digital hormone is sufficient to trigger and coordinate all actions of all r-cells.

Digital Hormones for Dynamic Communication. All communication among r-cells is accomplished by digital hormone propagation. We say that a r-cell *propagates* a digital hormone if it either generates a new digital hormone or it receives/sends an existing hormone. In the first case, the r-cell will send the new digital hormone to *all* active links. In the second case, the r-cell marks the receiving link and sends the digital hormone to the rest of its active links.

Propagation is the only way that r-cells communicate with each other. Different from regular message-passing protocols, a r-cell cannot arbitrarily select a subset of active links to send digital hormones. It must either propagate a digital hormone or completely ignore it. This property is similar to the communication protocols in artificial neural networks, but r-cells have dynamic links and an active link can both receive and propagate digital hormones. Digital hormones are also different from the conventional "broadcast" messages. Digital hormones are "propagated" through the network and they can be modified, delayed, relayed, or terminated by the receivers on the way of propagation, and there is no guarantee that all receivers will get the exact copy of the original digital hormone.

Digital hormone-based communication is also different from the concept of pheromone that recently gains much popularity in controlling multiple agent systems (Brueckner 2000). Digital hormones are signals that circulate inside a organism between cells, whereas pheromones are chemicals deposit externally in the environment and they are between organisms. Because of this difference, digital hormones and pheromones serve very different functions in biological systems.

Finally, communication based on digital hormones has a number of advantages over conventional communication methods. First, it can deal with the dynamic network problem because communication is accomplished by the bindings between a digital hormone and a receptor of the receiver r-cell. A sender does not need to know the names of the receivers, and parties of the network can come and go. A r-cell can migrate from one location to another yet the communication remains effective. This communication protocol is also efficient, for it can control complex global actions using a small number of signals, and scalable, for it puts no restriction on the internal structure of the communication parties nor their relationships.

Digital Hormones for Synchronization. Synchronization is such an important issue in self-organizational systems, and it is worthwhile to investigate how digital hormones accomplish that. The issue is to determine *when* r-cells should execute their actions so that desired global effects can be achieved. One approach to accomplish synchronization among r-cells is to designate a synchronization center that broadcasts synchronization signals to all subsystems. Such broadcasts can either be continuous or wait for the right moment. In either case, this approach must pay a high cost of communication because all subsystems must inform the center about their readiness. Another approach is to assume that every subsystem has a local clock that is synchronized globally. This approach requires no communication for synchronization but is not universally realistic.

In the digital hormone-based control framework, when the time for hormone propagation is negligible relative to the tasks to be synchronized, synchronization can be achieved naturally by the flexible interpretations of digital hormones. Since digital hormones can be "held" at a site for the occurrence of certain events before traveling further, they can be used as tokens for synchronizing events among r-cells. This is similar to the method of converge-cast discussed in (Lynch 1990) and this is especially good for serial synchronization.

The parallel synchronization requires more computational resources. A r-cell cannot execute its local action immediately after a hormone is received. It must negotiate with all r-cells in the party to ensure that all actions are started at the same time. For this purpose, we propose a synchronization algorithm, which runs on each r-cell in parallel and guarantees the same starting time (ignore the delay of hormone propagation) for all synchronized actions. The key idea is that, for any action that is triggered by a given digital hormone, a r-cell can infer that *all* other r-cells are ready to start when it receives the "expected number of digital hormones" from every active neighbor. Interestingly enough, the "expected number of digital hormones" for a neighbor is the number of active links of that neighbor. Thus, to achieve parallel synchronization, each r-cell in the group generates and propagates a hormone to all its neighbors before it executes its action. It then counts the number of received hormones for each active link. As soon as the expected number of hormones is received for a link, it generates a new hormone and propagate this new hormone to the rest of its neighbors. As soon as all active links have received the expected number of hormones, the r-cell executes the action and it is guaranteed to be parallel with all other r-cells' actions.

Digital Hormones for Collaboration. In a self-reconfiguration system, a r-cell can be triggered to generate a new digital hormone in two cases: by an external stimulation such as a visual signal or a radio command, or by a received digital hormone. Thus, there may be times when multiple digital hormones are simultaneously active in the organization. How do r-cells react to multiple hormones, especially when they are conflict to each other? How do conflicting digital hormones resolve themselves? These are the questions that must be answered by the management of digital hormones.

At the moment, we assume that each digital hormone carries a priority value initiated by its generator. When a r-cell generates a new digital hormone, it will associate a priority value to the hormone, and priority is determined by the nature of the stimuli or receptors. We assume that external stimuli will have higher priorities than the internal ones. For example, hormones triggered by the visual stimuli will have a higher priority than those triggered by audio stimuli, which in turn higher than those triggered by hunger. When a r-cell receives two conflicting hormones, it will response to the higher priority one and ignore the lower one. How a r-cell detects conflicts among multiple hormones is an interesting question. For initial implementation, we will give the conflict criteria to r-cells as constraints.

For a particular global task, the r-cell that generates the digital hormone is the temporal leader and coordinator of that task. For example, the tail r-cell is the leader for the task specified in an ALT digital hormone. Some actions are single events, and others may require a sequence of digital hormones. For a r-cell to generate a sequence of digital hormones, we assume that the r-cell has the local knowledge about the sequence. One possible representation of such knowledge is the action-list specified in a procedure. We assume that all r-cells are given the same initial knowledge about tasks, procedures, and primitive action lists. Thus any r-cell can become the leader of a task if it is required.

Digital hormones are terminated in a way similar to the way they are generated. That is, a r-cell can stop producing digital hormones either by self-promotion (because of an external stimulus) or by other digital hormones. For example, a digital hormone may be terminated if certain values are read from a local sensor, or if it receives a special digital hormone. For example, as shown in Figure 9, a r-cell will terminate its RCT when it receives a TAR.

Another interesting question about hormone management is how r-cells select actions to collaborate with one another. Since actions are selected independently by individual r-cells, it is a possible that actions of two r-cells violate some constraints in the gait procedure. Therefore a conflict resolution phase will also be required.

3.2 2001-2002 Period

To build self-reconfigurable systems and understand the principles of self-reconfiguration, we have been working on several joint projects with DARPA and AFOSR to develop prototypes and theories for physical self-reconfigurable systems. The CONRO project has built a metamorphic robot [1-6], the SOALA project [7-10] has developed a distributed control framework and related algorithms for self-reconfigurable systems, and the HORMCOMM project [11, 12] has developed a mathematical model for adaptive communication and self-organization in reconfigurable systems.

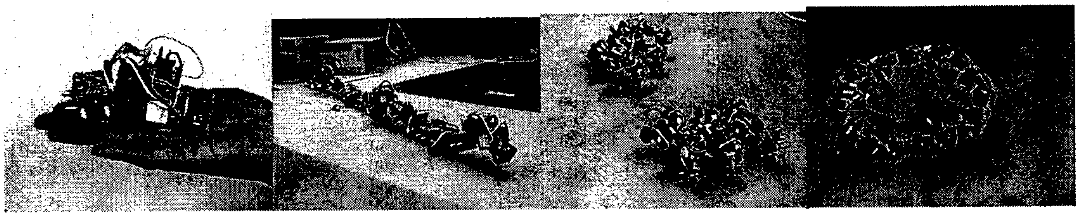


Figure 1: CONRO self-reconfigurable modules and configurations of snake, insects, and rolling track

The CONRO metamorphic robot is made from a homogeneous set of autonomous *reconfigurable modules* that can change their physical connections and configurations under computer or human command to meet the demands of the environment. Each CONRO reconfigurable module has a size of 1.0 inch^2 cross-section and 4.0 inch long and is equipped with a micro-controller, two servo motors, two batteries, four connectors for joining with other modules, and four pairs of infrared emitter/sensor for communication with neighboring modules and docking guidance. These modules are autonomous and self-sufficient and they have the docking capability and can connect and disconnect with each other to form various shapes. For example, Figure 1 shows a single module in a human hand, an 8-module snake, two 9-module six-legged insects, and an 8-module rolling track. Using novel control approaches, such as digital hormone model [1, 7-9] and role-based control [5, 13], developed in CONRO and SOALA, our reconfigurable robots can perform online *bifurcation, unification*, and *behavior-shifting* that are unique for reconfigurable systems. There is no fixed "brain" module in the CONRO robot, and every module behaves properly according to their relative position in the current configuration. For example, a moving CONRO snake robot can be bifurcated into pieces, yet each individual piece will "elect" a new head and continue to behave as an independent snake. Multiple snakes can be concatenated (for unification) while they are running and become a single and coherent snake. For online behavior shifting, a tail/spine module in a snake can be disconnected and reconnected to the side of the body, while the system is running, and its behavior will automatically change to a leg (the reverse process is also true). For fault tolerance, if a multiple legged robot loses some legs, the robot can still walk on the remaining legs without changing the control program. The CONRO robot can also perform self-reconfiguration in certain configurations. Figure 2 shows the steps of reconfiguration from a snake shape to a two-legged creature that can do a locomotion gait similar to the two-arm butterfly stroke in swimming. For movies and more information about CONRO, please visit the web site http://www.isi.edu/conro.

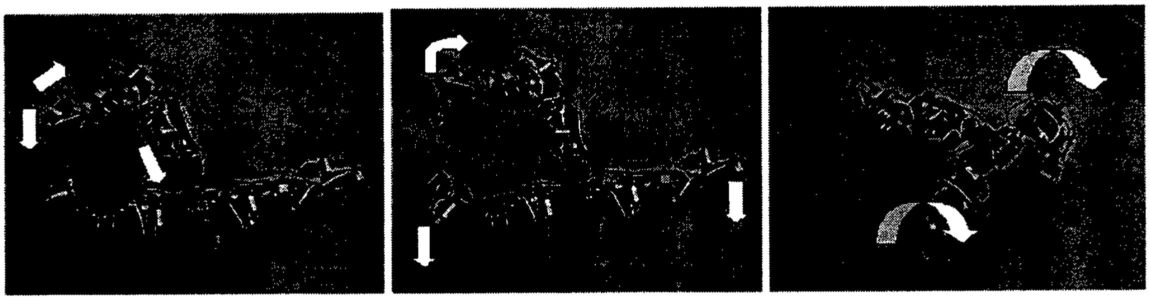


Figure 2: A CONRO "snake" self-reconfigures into a "butterfly walker."

It is a great challenge to control a self-reconfigurable system such as CONRO. Each module is an autonomous and intelligent agent, and its actions must cooperate with others in order to generate the desired global action in a given physical configuration. The concept of configuration can be interpreted in several ways. The physical interpretation is that it represents the structure or shape of the system. The connectivity interpretation is that it is a communication network topology. The control implication is that global actions (such as locomotion for a robot) require a re-computation of the local actions to be executed by the individual modules. These local actions depend on the position of the agent in the current configuration. In the past, we have focused on two general problems for self-reconfigurable systems: (1) how agents communicate with each other when connections and configurations change dynamically and unexpectedly, and (2) how agents collaborate local actions in the physically and tactically coupled organization. These two problems occur in many types of self-reconfigurable systems, including distributed sensor networks [14], swarm robotic system [15], and self-assembly system in space [16]. Note that ultimately the cooperative control in self-reconfigurable systems must be *dynamic*, to deal with the changes in network topology; *asynchronous*, to compensate the lack of global clocks; *scalable*, to support ever-growing structures and shape-alteration; *collaborative*, to enable global efforts by local actions in a physically and tactically coupled organization; *reliable*, to recover from local damages and provide fault-tolerance; and finally, *self-adaptation*, to select and form the best configuration for the task and environment in hand.

3.2.1 Adaptive Communication

As described above, the modules in a self-reconfigurable robot are reconfigured structurally. The physical interpretation of this action is that shape morphing occurs. The connectivity interpretation is that the modules have a new communication network topology. The control implication is that global actions such as locomotion require a re-computation of the local actions to be executed by the individual modules. These local actions depend on the position of the module in the reconfigured structure. To the best of our knowledge, such control approach can support some unique and new capabilities, such as distributed and online *bifurcation, unification*, and *behavior-shifting*, which have not been seen before in robotics literature. For example, a moving snake robot with many modules may be bifurcated into pieces, yet each individual piece can continue to behave as an independent snake. Multiple snakes can be concatenated (for unification) while they are running and become a single but longer snake. For behavior-shifting, a tail/spine module in a snake can be disconnected and reconnected to the side of the body, and its behavior will automatically change to a leg (the reverse process is also true). In fault tolerance, if a multiple legged robot loses some legs, the robot can still walk on the remaining legs without changing the control program. All these abilities would not be possible if modules could not cope with the topological changes in the communication network.

In this section, we describe an adaptive communication protocol for dynamic networks such as those used in self-reconfigurable robots. Using this protocol, modules can communicate even if the topology of the network is changing dynamically and unexpectedly. Communication with this protocol will be shown to be robust, flexible, and will allow reconfiguration while the network is in operation. The reconfiguration can either be self-initiated, superimposed by external agents, or in response to sensor interaction with the environment.

Using the concept of hormone messages and local topological types defined above, we can define the Adaptive Communication (AC) protocol for continual rediscovery of network topology and ensure adaptive communication. Figure 3 shows the pseudo-code program for the AC protocol. The main procedure is a loop of receiving and sending (propagating) "probe" hormones between neighbors, and selecting and executing local actions based on these messages. A probe is a special type of hormone that is used for continuously discovering and monitoring local topology. Other types of hormones that can trigger more actions will be introduced later. All modules in the network run the same program, and every module detects changes in its local topology (i.e., the changes in the active links) by sending probe messages to its connectors to discover if the connectors are active or not. The results of this discovery are maintained in the vector variable LINK[C], where C is the number of connectors for each module (e.g., C=4 for a CONRO module). If there is no active link on a connector *c* (or an existing active link on *c* is disconnected), then sending of a probe to *c* will fail and LINK[*c*] will be set to nil. If a new active link is just created through a connector *c*, then sending a probe to *c* will be successful and LINK[*c*] will be updated. After one exchange of probes between two neighbors, both sides will know which connector is involved in the new active link and their LINK variables will be set correctly[1].

```
OUT: the queue of messages to be sent out;
IN: the queue of messages received in the background;
C: the number of connectors for each module;
MaxClock: the max value for the local timer;
LINK[1,...,C]: the status variables for the connectors (i.e., the local topology), and their initially values are nil;
A hormone is a message of [type, data, sc, rc], where sc is the sending connector through which the message is sent, and rc is the receiving connector through which the message is received.

Main()
LocalTimer = 0;
Loop forever:
  For each connector c=1 to C, insert [probe,_,c,_] in OUT;
  For each received hormone [type, data, sc, rc] in IN, do:
     { LINK[rc] = sc;
       If (type ≠ probe) then
          SelectAndExecuteLocalActions(type, data);
          PropagateHormone(type, data, sc, rc);
     }
  Send();
  LocalTimer = mod(LocalTimer+1, MaxClock);
End Loop.

SelectAndExecuteLocalActions(type, data)
{ // For now, assume that when LocalTimer=0, a module will
  // generate a test hormone to propagate to the network
  // Other possible local actions will be introduced later.
  If LocalTimer==0, then for c=1 to C, do:
          Insert [Test, 0, c, nil] into OUT;
}

PropagateHormone(type, data, sc, rc)
{ For each connector c=1 to C, do:
    If LINK[c]≠0 and c≠rc, then
          { Delete [probe, *, c, *] from OUT;
            Insert [type, data, c, nil] into OUT; // propagation
          }
}

Send()
{ For each connector c=1 to C, do:
          get the first message [type,*,c,*] from OUT,
          Send the message through the connector c;
          If send fails (i.e., time out), LINK[c] = 0.
}
```

Figure 3. The Adaptive Communication (AC) Protocol

---

[1] For example, if an active link is created between the connector *x* of module A and the connector *y* of module B, then LINK[*x*]=*y* for module A, and LINK[*y*]=*x* for module B. The LINK[C] variable represents the local topology

The AC protocol has a number of important properties that are essential for adaptive communication in self-reconfigurable networks.

Proposition 1: Using the AC protocol, all modules can adapt to the dynamic topological changes in the self-reconfigurable network and discover their local topology in a time less than two cycles of the main loop. The updated local topology information is stored in LINK[*c*].

To see this proposition is true, notice that initially all LINK variables have a nil value. If a module has a neighbor on its connector *c*, then LINK[*c*] will be set properly when this module receives a probe on that connector. Since every module probes all its connectors in every cycle of the program, the LINK[*c*] will be updated correctly with at most two cycles.

Proposition 2: If the network is acyclic graph, then the AC protocol guarantees that every non-probe message will be propagated to every module in the network once and only once. The time for propagating a hormone to the entire network is linear to the radius of the network graph.

To see that proposition 2 is true, notice that when a new message is generated (e.g., [Test,*,*,*] in Figure 3), it will be sent to all active links from that module. When a module receives a hormone, it will send it to all active links except the link from which the hormone is received. Since the network is acyclic, the generator module can be viewed as the root of a propagation tree, where each module will receive the hormone from its parent, and will send the hormone to all its children. The propagation will terminate at the leaf nodes (modules) where there is no active links to propagate. Since the tree includes every module, the hormone reaches every node. Since every module in the tree has only one parent, the hormone will be received only once by any module.

For networks that contain loops (cyclic graphs), the AC protocol must be extended to prevent a hormone from propagating to the same module again and again. To ensure that each hormone is received once and only once by every module, additional local information (such as local variables) must be used to "break" the loop of communication. We will illustrate the idea in the ADC protocol when we describe the control of rolling tracks, which is a cyclic network.

3.2.2 Hormone-Inspired Distributed Control

As described above we want a distributed control protocol that is identity free but supports a module to select its actions based on its location in the network. Since hormones can trigger different actions at different site and every module continuously discoveries its local topology, such a control method can be defined based on the hormone messages.

To illustrate the idea, let us first consider an example of how hormones are used to control the locomotion of a metamorphic snake robot. Consider a 6-module CONRO snake robot and its caterpillar gait. The types of modules, from the left to the right, in this robot are: T1 (the head), T16, T16, T16, T16, and T2 (the tail). To move forward, each module's pitch motor (DOF1) goes through a series of positions and the synchronized global effect of these local motions is a forward movement of the whole caterpillar (indicated by the arrow). In general, the wavelength of the gait can be flexible (e.g., a single module can craw as a caterpillar).

To completely specify this gait, one can use a conventional gait control table [6] shown, where each row in the table corresponds to the target DOF1 positions for all modules in the configuration during a step. Each column corresponds to the sequence of desired positions for one DOF1. The control starts out at the first step in the table, and then switches to the next step

---

type of a CONRO module. For example, a module is type T0 if LINK[f,l,r,b] = [nil,nil,nil,nil]; type T2 if LINK[f,l,r,b] = [b,nil,nil,nil]; and type T21 if LINK[f,l,r,b] = [nil,b,b,f].

when all DOF1 have reached their target position in the current step. When the last step in the table is done, the control starts over again at step 0.

The problem of this conventional gait table method is that it is not designed to deal with the dynamic nature of robot configuration. Every time the configuration is changed, no matter how slight the modification is, the control table must be rewritten. For example, if two snakes join together to become one, a new control table must be designed from scratch. A simple concatenation of the existing tables may not be appropriate because their steps may mismatch. Furthermore, when robots are moving on rough ground, actions on each DOF cannot be determined at the outset.

To represent a locomotion gait using the hormone idea, we notice that Table 2 has a "shifting" pattern among the actions performed by the modules. The action performed by a module *m* at step *t* is the action to be performed by the module (*m*-1) at step (*t*+1). Thus, instead of maintaining the entire control table, this gait is represented and distributed at each module as a sequence of motor actions (+45°, -45°, -45°, +45°). If a module is performing this caterpillar gait, it must select and execute one of these actions in a way that is synchronized and consistent with its neighbor module. To coordinate the actions among modules, a hormone can be used to propagate through the snake and allow each module to inform its immediate neighbor what action it has selected so the neighbor can select the appropriate action and continue the hormone propagation. This example also illustrates that hormones are different from broadcasting messages because their contents are changing during the propagation.

To implement the hormone-inspired distributed control on the AC protocol, each module must react to the received hormones with appropriate local actions. These actions include the commands to local sensors and actuators, updates of local state variables, as well as modification of existing hormones or generation of new hormones. Modules determine their actions based on the received hormone messages, their local knowledge and information, such as neighborhood topology (module types) or the states of local sensors and actuators.

```
// Built on the AC protocol by adding a RULEBASE and
// extending the following procedure.

SelectAndExecuteLocalActions(type, data)
{ // Select appropriate actions based on
  // type, data, LINK, LocalTimer, and RULEBASE;

  Actions←SelectActions(type,data,LINK,LocalTimer,RULEBASE);
  For each action a in Actions, do ExecuteAction(a);
}

RULEBASE:
{ // The rules here are similar to the receptors in biological cells.
  // They are task-specific "if-then" rules as those in Table 3;
  // Although each desired task has a different set of rules,
  // the rules can be combined together if they are not conflicting.
}
```

Figure 4. The Adaptive and Distributed Control Protocol

For these purpose, we specify the Adaptive and Distributed Control (ADC) protocol listed in Figure 4. The ADC protocol is the same as the AC protocol except that there is a RULEBASE and the procedure SelectAndExecuteLocalActions() is extended to select and execute actions based on the rules in the RULEBASE. The selection process is based on (1) local topology information (such as LINK[] and the module type), (2) the local state information (such as local timer, motor and sensor states), and (3) the received hormone messages. Biologically speaking, the rules in RULEBASE are analogous to the receptors in biological cells, which determine when and how to react incoming hormones. A module can generate new hormones when triggered by the external stimuli (e.g., the environmental features such as color or sound) or by a received hormone message. When there are multiple active hormones in the system, the modules will negotiate and settle on one hormone activity.

To illustrate the idea of action selection based on rules, let us consider how the caterpillar movement is implemented. The required rules for this global behavior are listed in Table 3. In this table, the type of the hormone message is called CP, and the data field contains the code for DOF1. The other fields of hormones are as usual, but we only show the field of sender connector (*sc*) for simplicity.

TABLE 3: THE RULEBASE FOR THE CATERPILLAR MOVE

Module Type	Local Timer	Received Hormone Data	Perform Action	Send Hormone
T1	0		DOF1=+45	[CP, A, b]
T1	(1/4)*MaxClock		DOF1=-45	[CP, B, b]
T1	(1/2)*MaxClock		DOF1=-45	[CP, C, b]
T1	(3/4)*MaxClock		DOF1=+45	[CP, D, b]
T16,T2		A	DOF1=-45	[CP, B, b]
T16,T2		B	DOF1=-45	[CP, C, b]
T16,T2		C	DOF1=+45	[CP, D, b]
T16,T2		D	DOF1=+45	[CP, A, b]

All modules in the robot have the same set of rules, but they react to hormones differently because each module has different local topology and state information. For example, the first four rules will trigger the head module (type T1) to generate and send (through the back connector *b*) four new hormones in every cycle of MaxClock, but will have no effects on other modules. The last four rules will not affect the head module, but will cause all the body modules (T16) to propagate hormones and select actions. These modules will receive hormones through the front connector *f* and propagate hormones through the back connector *b*. When a hormone reaches the tail module (T2), the propagation will stop because the tail module's back connector is not active. The speed of the caterpillar movement is determined by the value of MaxClock. The smaller the value is, the more frequent new hormones will be generated, thus faster the caterpillar moves.

Compared to the gait control table, the ADC protocol has a number of advantages. First, it supports online reconfiguration and is robust to a class of shape alterations. For example, when a snake is cut into two segments, the two disconnected modules will quickly change their types from T16 to T2, and from T16 to T1, respectively (due to the AC protocol). The new T1 module will serve as the head of the second segment, and the new T2 module will become the tail of the first segment. Both segments will continue move as caterpillar. Similarly, when two or more snakes are concatenated together, all the modules that are connected will become T16, and the new snake will have one head and one tail, and the caterpillar move will continue with the long snake. Other advantages of this hormone-inspired distributed control protocol include the scalability (the control will function regardless of how many modules are in the snake configuration) and the efficiency (the coordination between modules requires only one hormone to propagate from the head to the tail). Let n be the number of modules in the snake, then the ADC protocol requires only $O(n)$ message hops for each caterpillar step, while a centralized approach would require $O(n^2)$ message hops because n messages must be sent to n modules.

In general, the ADC protocol has the following properties:

1. Distributed and Fault-Tolerant. There are no permanent "brain" modules in the system and any module can dynamically become a leader when the local topology is appropriate. Damage to single modules will not paralyze the entire system.
2. Collaborative Behaviors. Modules do not require unique IDs yet can determine their behaviors based on their topology types and other local information. The global behaviors can be locomotion or self-reconfiguration.

3. Asynchronous Coordination. No centralized global real time clocks are needed for module coordination, and actions can be synchronized via hormone propagation.
4. Scalability. The control mechanism is robust to changes in configuration as modules can be added, deleted, or rearranged in the network.

The ADC protocol can be applied to many different robot configurations. All that is required is to provide the appropriate set of rules to the protocol and have the correct initial configuration in place. For example, Table 4 lists the set of rules that will enable a legged robot to walk. In other words, the left leg modules are T6, the right leg modules are T5, the head is T21, the tail T19, and the spine modules are T29. The hormone message used in Table 4 is named as LG. We use set notation such as {l,b,r} as a shorthand for the set of connectors to send the hormone. The action Straight means DOF1=DOF2=0. The action Swing means to lift a leg module, swing the module forward, and then put the module down on the ground. The action Holding means to hold a leg module on the ground while rotating the hip to compensate the swing actions of other legs.

TABLE 4: THE RULEBASE FOR A LEGGED WALK

Module Type	Local Timer	Received Hormone Data	Perform Action	Send Hormone
T21, T17, T18	0		Straight	[LG, A, {l,r,b}]
T21, T17, T18	0.5*MaxClock		Straight	[LG, B, {l,r,b}]
T29, T19, T26, T28		A	Straight	[LG, B, {l,r,b}]
T29, T19, T26, T28		B	Straight	[LG, A, {l,r,b}]
T5		A	Swing	
T6		B	Holding	

The first two rules indicate that the head module, which can be type T21, T17, or T18, is to generate two new LG hormones with alternative data (A and B) for every cycle of MaxClock. This hormone propagates through the body modules (T29, T26, or T28) and the tail module (T19), alternates its data field, and reaches the leg modules, which will determine their actions based on their types (T5 or T6).

This control mechanism is robust to changes in configurations. For example, one can dynamically add or delete legs from this robot, and the control will be intact. The speed of this gait can be controlled by the value of MaxClock, which determines the frequency of hormone generation from the head module.

As another example of how to use the ADC protocol to control locomotion of self-reconfigurable robots, Figure 5 shows the configuration of the rolling track. Notice that in this configuration, all modules are of type T16, only their DOF1 values are different. The track moves one direction by shifting the two DOF1 values (90, 90) to the opposite direction.

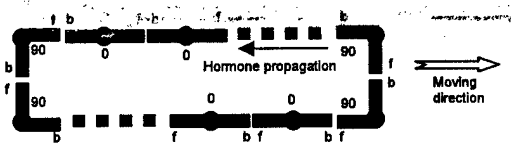


Figure 5. A rolling track configuration.

Table 5: The RULEBASE for a rolling track

Module Type	Local Variables	Received CP Data	Perform Action	Send Hormone
T16	Head=1, Timer=MaxClock		DOF1=90, Timer=0, Head=0	[RL,(90,90,1),b]
T16	DOF1=0	(90,90,1)	DOF1=90, Timer=0, Head=1	[RL,(90,0,0),b]*
T16	DOF1=0	(90,90,0)	DOF1=90, Timer=0, Head=0	[RL,(90,0,0),b]
T16	DOF1=0	(90,0,0)	DOF1=0, Timer=0, Head=0	[RL,(0,0,0),b]
T16	DOF1=0	(0,0,0)	DOF1=0, Timer=0, Head=0	[RL,(0,0,0),b]
T16	DOF1=90	(0,0,0)	DOF1=0, Timer=0, Head=0	[RL,(0,90,0),b]
T16	Head=0, DOF1=90	(0,90,0)	DOF1=90, Timer=0, Head=0	[RL,(90,90,0),b]
T16	Head=1, DOF1=90	(0,90,0)	DOF1=90, Timer=0, Head=0	[RL,(90,90,1),b]

Note: * means send the hormone after all local actions are completed.

Table 5 lists the rules for a rolling track robot. The hormone used here is of type RL, and its data field contains two values of DOF1, and a binary value for selecting the head module. One hormone message continuously propagates in the loop (just as a token traveling in a token ring) and triggers the modules to bend (DOF1=90) or straighten (DOF1=0) in sequence. We assume that there is one and only one module whose local variable Head=1. This module is responsible for generating a new hormone when there is no hormone in the loop. This is implemented by the first rule, which will detect a time-out for not receiving any hormone for a long time (i.e., looping through the program for MaxClock times). The head module is not fixed but moving in the loop. We assume that the initial bending pattern of the loop is correct (i.e., as shown in Fig. 8) and the head module is initially located at the up-right corner of the loop. The rules in Table 5 will shift the bending pattern and the head position in the loop and cause the loop to roll into the opposite direction of hormone propagation. Since hormone propagation is much faster than the actual execution of actions, when a module is becoming the head, it is also responsible for making sure all actions in the loop are completed before the next round starts. The head module will hold the next hormone propagation until all its local actions (DOF1 moving from 0 to 90) are completed.

Notice that the loop configuration is a cyclic network and module types alone are no longer sufficient to determine local actions (in fact all modules in the loop have the same type T16). In general, additional local variables (such as Head) are necessary to ensure the global collaborations between modules in a cyclic network.

Due to the potential of communication errors, there may be situations where no module has the local variable Head=1 and there is a need for a new head module. In such a case, it may be possible to create a negotiation mechanism for one module to switch its local variable to Head=1, if there are none in the group -- just like some schools of fish where a female changes gender if the male in the group is dead. One possible implementation is to allow any module to self-promote to become a new head if it has not received messages for a long time. In this case, modules must negotiate among to ensure that there is one and only one head in the system.

3.2.3 Experimental Results

The hormone-inspired adaptive communication and distributed control algorithms described above have been implemented and tested in two sets of experiments. The first is to apply the algorithm to the real CONRO modules for locomotion and reconfiguration. The second is to apply the algorithm to a CONRO-like robot in a Newtonian mechanics simulation environment called Working Model 3D [39].

All modules are loaded with the same program that implements the ADC protocol illustrated in Figure 3 and Figure 4. For different configurations, we have loaded the different RULEBASE.

All modules are running as autonomous systems without any off-line computational resources. For economic reasons, the power of the modules is supplied independently through cables from an off-board power supplier.

For the snake configuration, we have loaded the rules in Table 3 onto the modules and experimented with caterpillar movement with different lengths ranging from 1 module to 10 modules. With no modification of programs, all these configurations can move and snakes with more than 3 modules can move properly as caterpillar. The average speed of the caterpillar movements is approximately 30cm/minute. To test the ability of on-line reconfiguration, we have dynamically "cut" a 10-module running snake into three segments with lengths of 4, 4, and 2, respectively. All these segments adapt to the new configuration and continue to move as independent caterpillars. We also dynamically connected two or three independent running caterpillars with various lengths into a single and longer caterpillar. The new and longer caterpillar would adapt to the new configuration and continue to move in the caterpillar gait. These experiments show that the ADC protocol is robust to changes in the length of the snake configuration.

To test whether modules can automatically generate hormones when they receive appropriate environmental stimuli from their local external sensors, we have installed two tilt-sensors on one of the modules in the snake configuration, and loaded the following rules to the modules:

If tilt-sensors=[0,1], generate hormone [FlipLeft,*,*]
If tilt-sensors=[1,0], generate hormone [FlipRight,*,*]
If tilt-sensors=[1,1], generate hormone [FlipOver,*,*]

We defined the actions for FlipLeft, FlipRight, and FlipOver for all the modules so that when these hormone messages are received, the modules will perform the correct actions for DOF1 and DOF2 to flip the snake back to its normal orientation. To test this new behavior, we manually pushed the snake, while it is moving as a caterpillar, to its side or flipped it upside down. We observed that the tilt-sensors are activated, new hormones are generated, a sequence of actions is triggered, and the robot flips back to its correct orientation. (See movies at http://www.isi.edu/conro.)

For the legged configuration, we have loaded the rules in Table 4 onto the modules and experimented with the various configurations derived from a 6-leg robot (see Fig. 3). These configurations can walk on different number of legs without changing the program and the rules. While a 6-leg robot is walking, we dynamically removed one leg from the robot and the robot can continue walk on the remaining legs. The removed leg can be any of the 6 legs. We then dynamically removed a pair of legs (the front, the middle, and the rear) from the robot, and observed that the robot can continue walk on the remaining 4 legs. We then systematically experimented removing 2, 3, 4, 5, and 6 legs from the robot, and observed that the robot would still walk if the remaining legs can support the body. In other cases, the robot would still attempt to walk on the remaining legs even if it has only one leg. Although we have only experimented robots with up to 6 legs, we believe in general these results can scale up to large configurations such as centipedes that have many legs.

For the rolling track configuration, we have loaded the rules in Table 5 onto the modules and experimented with rolling tracks with lengths of 8, 10, and 12. In all these configurations, the rolling track moved successfully with speed approximately 60cm/minute. The current configurations must have more than 6 modules and the number of modules must be even. This is because there must be 4 modules with DOF1=90, and at least two other modules with DOF1=0. To test the robustness of the system against loss of messages in the communication, we simulated random message losses in the program. We observed that when a message of [RL, (*,*,0), b] is lost, the robot will stop rolling momentarily and then the head module's local timer will reach MaxClock, and a new hormone will be generated and the track will resume rolling. If the lost message is [RL, (*,*,1), b], then there will be no head module in the system, and the robot will not roll again. However, since most messages are of the first kind, the chance of failing to resume rolling is low. In practice, when message losses do occur, we only observed non-recovery stops in rare occasions.

In parallel with the experiments on the real CONRO robot, we have also implemented with the ADC protocol on a simulated CONRO-like robot in a software Newtonian simulation environment called Working Model 3D [39]. Using this three-dimensional dynamics simulation program, we have designed a set of virtual CONRO modules to approximate the physical properties of the real modules, including their mass, motor torques, joints, coefficient of friction, moments of inertia, velocities, springs, and dampers. The ADC protocol is implemented in Java and runs on each simulated module. We have experimented with and demonstrated successful locomotion in various configurations, including snakes with different length (3-12 modules) and insects with different numbers (4-6) of legs.

3.3 2002-2003 Period

The construction and control of self-reconfigurable systems is a very challenging problem. A self-reconfigurable system must adapt not only its behaviors by learning from experience, but also its configurations by the needs of the mission and the environment in hand. Instead of a single intelligent entity, a reconfigurable system is an organization of many intelligent, autonomous, cooperative nodes/agents/robots that have both physical and logical connectivity. A self-reconfigurable system is by definition multifunctional, robust, flexible, adaptive, and capable of maintaining a long-term uninterrupted operation and dealing with situations that were not anticipated by the designers. The control of self-reconfigurable systems require mental abilities that are equivalent to human-level intelligence, and physical abilities that are equal or superior to that of biological systems.

Self-reconfigurable systems are inherently distributed. During a problem solving process, new agents might join and faulty agents may leave the network at any given time. As a result, the number of agents in a self-reconfigurable system and its topology are continuously changing. Agents are autonomous entities without unique global identifiers and their control processes are running asynchronously. This means that they may execute actions and/or communicate with other agents at anytime. In addition, they can only communicate with their neighboring agents and have limited information about the network. Such characteristics make the problem of controlling self-reconfigurable systems a challenging problem.

3.3.1 Distributed Task Negotiation

The goal in controlling a self-reconfigurable system (SRS) is to accomplish a given task. A SRS can accomplish a task by generating a suitable group behavior for the given task. However, agents in a SRS are autonomous entities and each agent is capable of initiating multiple tasks. Therefore it is possible that multiple agents initiate many tasks at a given time. Many of these tasks might be competing or even conflicting with each other. In such situation, the first step for all agents in solving the problem of controlling a SRS is to negotiate on selecting the same task.

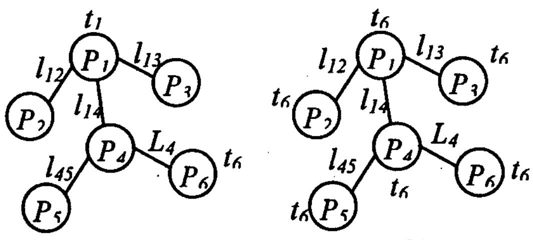
This is a very challenging problem due to several reasons: the relationships among agents are not static but change with the configuration of the network, agents do not have unique global identifiers or addresses, agents do not know the global configuration in advance, and can only communicate with immediate neighbors.

The previous approaches for solving the problem of distributed task negotiation can be divided into two categories. The first category includes the centralized approaches. They assume that there is a central designated agent that selects a single task among the initiated tasks in the network for all agents. For example, in the controller of the robot Toto [19] the *active node*, which represents the current position of the robot, is the only node that does the selection. However, such centralized approaches are inefficient for two reasons: 1) the central controller agent creates a single-point failure for the entire system and 2) these approaches do not scale well. Because communicating with the central controller agent becomes a bottleneck for the rest of the agents. Other non-centralized approaches for solving this problem such as role-based control [20] avoid the situation by assuming that there is only one agent can be the task initiator. However, such an assumption constrains the robots capabilities in many real-world problems, where multiple tasks might be generated.

The distributed task negotiation problem is a prominent problem in self-reconfigurable systems such as self-reconfigurable robots, sensor networks, and multi-agent organizations. This problem arises in situations where the autonomous agents in the self-reconfigurable system initiate multiple tasks. These tasks might be competing or even conflicting with each other. For example, in a snake shape self-reconfigurable robot, the tail module may want to move forward, while the head module may want to avoid an obstacle by moving left. Therefore, selecting the right task when there are many competing choices is a critical problem for controlling the self-reconfigurable systems.

Formally, a distributed task negotiation problem consists of a tuple (P, L, T, S), where P is a list of agents, p_i, such that $i \in \{1, \ldots, N\}$; L is a list of communication links, l_{jk}, such that $j,k \in \{1, \ldots, N\}$; T is a list of tasks, t_m, such that $1 \leq m \leq N$, and S is a set of task selection functions, S_i: $(T') \rightarrow t_i$, such that $i \in \{1, \ldots, N\}$ and $T' \subset T$. T' is called the available tasks. The task selection function of each agent selects a single task from the set of the available tasks. Note that the size of the network is unknown to the individual agents and the index number, i, assigned to each agent is only used for defining the problem and not used in the negotiation process. A distributed task negotiation problem is solved when all agents have selected the same task from T, called t^*, and have been notified that the negotiation process is terminated.

To illustrate the above definition, consider the example in Figure 1a, where $P = \{p_1, p_2, p_3, p_4, p_5, p_6\}$, $L = \{l_{12}, l_{14}, l_{13}, l_{45}, l_{46}\}$, $T = \{t_1, t_6\}$, and S is a selection function that prefers tasks with larger indexes and shared by all nodes. Initially, agents p_1 and p_6 have initiated two tasks, t_1 and t_6, respectively, and the rest of the agents are waiting to receive tasks. Figure 1b shows a solution for the given problem where all agents have selected task t_6.



(a) (b)

Figure 1: An example of a distributed task selection problem. a) Initially p_1 and p_6 initiated two tasks (t_1, t_6). b) A solution, when all agents have selected $t^* = t_6$.

The distributed task negotiation problem occurs in many types of distributed systems, including for example, sensor network [Estrin, 1999 #73], swarm robots [Bonabeau, 1999 #74], or multi-agent systems [Shen, 2002 #54]. In distributed multi-robot systems, previous approaches such as the 'role-based' approach [Stoy, 2002 #70] prevent initiation of multiple tasks and allow only one agent to be the task initiator in the entire network. Other approaches such as [Mataric, 1992 #63] [Yim, 1994 #1] allow all agents to initiate multiple tasks but assume a designated central agent selects and dictates a task to all conflicting agents. Another field that faces the same problem is distributed computing and algorithm design [Lynch, 2000 #43].

This work is different from all existing approaches. Unlike centralized approaches, DISTINCT algorithm scales well with configurations and is robust to individual module failures. By letting all modules to be the task initiators, DISTINCT allows cooperative distributed problem solving [Durfee, 1991 #76] and can deal with both task negotiation and termination detection.

Assigning priorities to the competing tasks and forcing the nodes to select tasks that have higher priorities [Lynch, 2000 #43] will not solve this problem since the number of nodes and initiated tasks in the network are unknown. Additionally, in situations where the priority of the tasks can change during the negotiation process, it is extremely difficult to determine the correct priorities for an arbitrary set of competing tasks.

In centralized approaches, tasks do not need priorities. Instead, all initiated tasks are sent to a central agent and this agent broadcasts its selected task to the rest of the agents. The limitations of the centralized approaches are that they are not scalable and that communication with the central agent is a bottleneck in the network. In addition, the failure of the central agent disables the entire system.

In our solution, agents propagate their tasks to their neighbors. Meanwhile, agents generate a Task Spanning Tree (TST) for each propagated task. As a result, when more than one task is initiated, a forest of partial TSTs is created. These partial TSTs negotiate with each other and gradually merge into one and only one TST. This final TST represents the task that has been selected by all agents in the network. During the tree building process, all agents report their status to their parent agents. The negotiation process terminates when an agent that has no parent has received reports from all of its children. This agent is the root of the final TST, and it then notifies all agents in the tree with an "end of task negotiation" message and all agents will select the task associated with the final TST.

For agents that have competing tasks to select a single task, the goal is to create a single TST. Each agent must decide on two issues: 1) what task to select and propagate, and 2) how to be a part of a TST.

Initially, task-initiating agents propagate their tasks by propagating *task messages* (*TM*) to their neighboring agents and designating themselves as the root of a partial TST. Assuming that the recipients of these messages are non-initiating agents and that each agent receives only one *TM*, the recipients of the *TM* adopt the received task and create a "child-of" relationship toward the sender of the *TM*. The non-initiating agents will in turn propagate the received task by sending new *TM* to the rest of their neighbors, excluding their parent.

To illustrate the idea, Figure 2 shows an example in which, agents P_1 and P_6 are the initiators of tasks t_1 and t_6 respectively and the rest of the agents are non- initiator agents. Agents P_2 and P_3 are the recipients of the *TMs* propagated by P_1, $TM(t_1)$, and therefore have selected task t_1. Similarly, P_4 and P_5 are the recipients of the *TM* propagated by P_6, $TM(t_6)$, and therefore have selected task t_6. In this situation, parallel arrows show the "child-of" relationships that the agents have created.

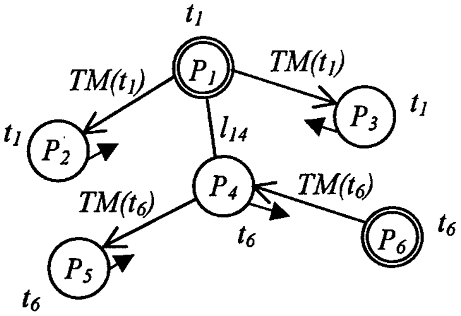


Figure 2: Task message propagation. Arrows on the links indicate messages in transit and arrows parallel to links indicate the "child-of" relationship. Double circles indicate the roots of partial TSTs.

Based on the above assumption, no message has been propagated through the link l_{14}. As a result two TSTs have been formed; one rooted at P_1 and the other rooted at P_6. In each TST, all agents have selected the same task.

At this point, if we relax the above assumption at this point, two cases might occur. **1)** either a root agent receives a *TM*, or **2)** a non-root agent receives a *TM* from an agent that is not its parent. An Example of the first case happens in Figure 2 when P_1, a root agent, receives a *task message* from P_4 and an example of second case happens when P_4, a non-root agent in the TST rooted at P_6, receives a *TM* from P_1, which belongs to another partial TST.

In the first case, the recipient, which is a root agent, drops being a root, adopts the received task, establishes a "child-of" relationship with the sender of the *TM*, and propagates new *TMs* to the rest of its neighbors, which are its children. In this situation, these agents adopt the new received task and propagate it to the rest of their neighbors.

In the second case, the received *TM* is a conflicting message since it was received from a non-parent agent. To resolve the conflict, the recipient agent detect deletes all of its previous "child-of" relationships, makes a choice between the received and previously selected task (using its task selection function), propagates *newRoot messages* (*NRM*) containing the new selected task to all of its neighbors, and then designates itself as a new root for the selected task.

The role of the *NRM* is to merge partial TSTs and create a new root for the resultig TST. Therefore, the recipient of a *NRM* adopts the received task, creates a new "child-of" relationship with the sender of the *NRM*, becomes a non-root agent (if previously a root), and propagates a new *NRM* containing the received task to the rest of its children

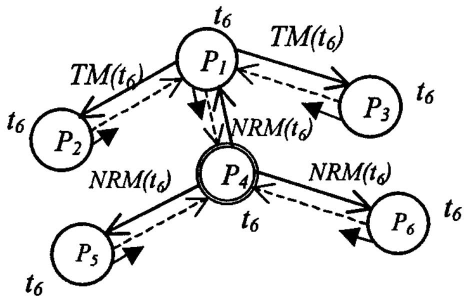


Figure 3: Merged partial TSTs shown in Figure 2.

P_4 is the new root of the merged TSTs. Dashed arrows indicate the ack messages.

Figure 3 shows the result of merging the two partial TSTs in Figure 2 for the situation where P_4 has been the agent that has received a conflicting *TM* from P_1. As a result, P_4 selects t_6 between t_6 and t_1 (task with the higher index), promotes itself to be the root of the new TST, and propagates $NRM(t_6)$ to P_1, P_5, and P_6, which turns P_1, and P_6 to non-root agents. Consequently, P_1 will adopt t_6 as its new task and propagate a new to P_2 and P_3. This *TM* message causes task switch in those agents. As shown in Figure 3, the final result of the task negotiation process is a single TST with a specified root and a selected task. However, at this point the agents do not know that the task negotiation process has been terminated. Unless a mechanism for detecting the termination of the negotiation is in place, the agents would wait indefinitely. For this purpose, we have developed a distributed termination detection mechanism, which is based on communicating *Acknowledgement messages (AM)* with the parent agent for the received *TM* and *NRM* messages.

The Distributed task negotiation process described above has been implemented as an algorithm called DISTINCT. Given a distributed task negotiation problem, this algorithm ensures that all agents select the same task coherently regardless of the number of competing tasks initiated in the network. Figure 4 illustrates the high-level description of the DISTINCT algorithm.

Four types of messages are used. First, a *task message* (*TM*) is used for propagating the initiated tasks. Second, a *newRoot message* (*NRM*) is propagated when a conflict is detected and partial TSTs are to be merged. Third, an *ack messages* (*AM*) is used for detecting the termination event. Finally, a *taskSelected message* is propagated from the root of the final TST to all nodes in the network.

Task initiator agents begin by calling the **initiated** procedure then wait for incoming messages. The 'Links' variable is the list of the communication links of an agent. In addition, the *ParentLink* and *ChildLinks* variables specify the parent-child relationships among agents in a TST. In line (a) of the **initiated** procedure, an agent designates itself as a root agent by assigning a **null** value to its *ParentLink* variable. As a result, all of the communication links (Links) of the root agents are marked as *ChildLinks*.

```
when initiated (task (t)) do
  SelectedTask = t;
  ParentLink = null;                    (a)
  ChildLinks = Links
  for each $L \in$ ChildLinks do
    L.ackProcessed = false;
  end do;
  for each $L \in$ ChildLinks do
    send (L ,task (t))
  end do;
end do;
when received (task (t), link (j)) do
  for each $L \in$ Links do
   L.ackProcessed = false;
  end do;
  if (SelectedTask = null or ParentLink = j)
    SelectedTask = t;
   ParentLink = j;
   ChildLinks = Links - j;
   if (ChildLinks is not empty)
      for each $L \in$ ChildLinks do
        send (L ,task (t));
      end do;
   else send (ParentLink, ack (t)); end if;
  else SelectedTask =SelectionFunction (t, SelectedTask);
   ParentLink = null;
   ChildLinks = Links
   for each $L \in$ ChildLinks do
     send (L ,newRoot (SelectedTask))
    end do; end if;
end do;
when received (newRoot (t), link (j)) do
  SelectedTask = t;
 ParentLink = j;
  for each $L \in$ Links do
   L.ackProcessed = false;
  end do;
  ChildLinks = Links - j;
  if (ChildLinks is not empty)
   for each $L \in$ ChildLinks do
     send (L ,newRoot (t)) end if; end do;
  else send (j, ack (t)); end if;
end do;
when received (ack (t), link (j)) do
  j. ackProcessed = true;
  acknowledgeComplete? = true;
  for each $L \in$ ChildLinks do
   if (L. ackProcessed =false)
     acknowledgeComplete? = false;
     break; end if; end do;
  if (acknowledgeComplete? = true)
   if (ParentLink $\neq$ null)
     ParentLink. ackProcessed = true;
     send (ParentLink, ack (t));
   else send(Links, taskSelected (t)); end if; end if;
end do;
```

Figure 4: The DISTINCT Algorithm

The *ackProcessed* variable is used for keeping track of the received *ack messages* to detect the task negotiation termination event. The currently selected task is stored in the *SelectedTask* variable. The *acknowledgeComplete*? in the **ack** procedure is a local variable that checks if all the expected number of **ack** messages are received. When the value of this local variable is true for the root of a TST, it detects that a single TST has been formed and the task negotiation process is terminated. Consequently, it propagates a **taskSelected** message to all of its children. The recipients of these messages will call the **taskSelected** procedure and eventually all agents in the network will select the same task and the negotiation process is successfully terminated.

We showed that the DISTINCT algorithm will reach a stable state when all agents have selected the same task. Assume there are N agents in the network. As a result of communication of the initiated tasks, and just before any conflict is detected, the network is partitioned into a set of non-overlapping sub-trees, which are the partial TSTs. Agents in the same partial TST have selected the same task.

Based on the property that any two nodes in the tree are connected by a unique path, we may conclude that there is at most one connecting link between any two partial TSTs. Otherwise there will be more than one path from a node in one partial TST to a node in the other partial TST, which will contradict the above-mentioned property. Consequently, if each partial TST is considered to be a single "super" node, the resulting network is also a tree; see Figure 5. The connecting links of these nodes are called *conflicting links* since the messages that they transfer cause conflicts in the recipient agents.

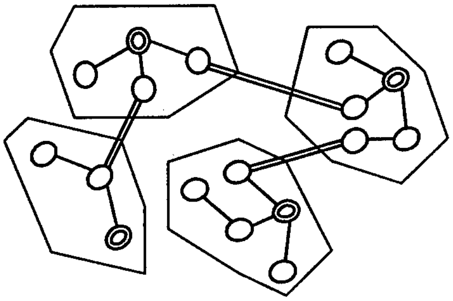


Figure 5: A task network is partitioned by partial TSTs. Nodes are surrounded by polygons and double lines indicate the conflicting links.

Based on the above description, and by considering that this algorithm merges partial TSTs that have conflicting links between them, DISTINCT algorithm will eventually produce a single TST. Furthermore, since the selected task for all merged TSTs is the same, only one task will be selected. In addition, due to the facts that there are only N-1 links (vertices) in a connected tree with N nodes, and that merging will monotonically reduce the number of nodes, the number of *conflicting links* will monotonically reduce to zero. This means a single TST can be created after at most N times merging.

The complexity of DISTINCT can be estimated as follows. In the worst case, every initiated task may override all of the other nodes selected tasks, therefore the worst-case time complexity of the DISTINCT algorithm is O(NT) where N is the number of nodes and T is the number of initiated tasks. Similarly, the total number of communicated messages is at most NT. It means that each task is communicated to at most N nodes. If we assume that each node requires receiving at least one message to know about a newly initiated task, we can conclude that the number of messages communicated by DISTINCT algorithm is optimal.

In our previous work for the distributed control of locomotion and reconfiguration, [3], we assumed that only one task was generated by one module in the robot at a time. In the presence of the DISTINCT algorithm, we can now relax this assumption.

Using the DISTINCT algorithm, a CONRO robot can select a single task among multiple initiated tasks. For example, Figure 6 shows the schematic view of a four-legged CONRO robot and its equivalent agent network. Two modules of the CONRO robot have initiated *forward walk* and *Obstacle Avoidance* tasks. Therefore, as we saw earlier, the robot is capable of selecting a single task and detecting the task negotiation termination event. In this experiment, *Obstacle Avoidance* had a higher priority than the *forward walk* task.

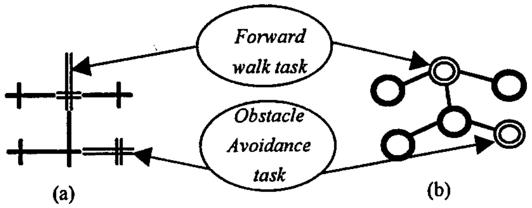


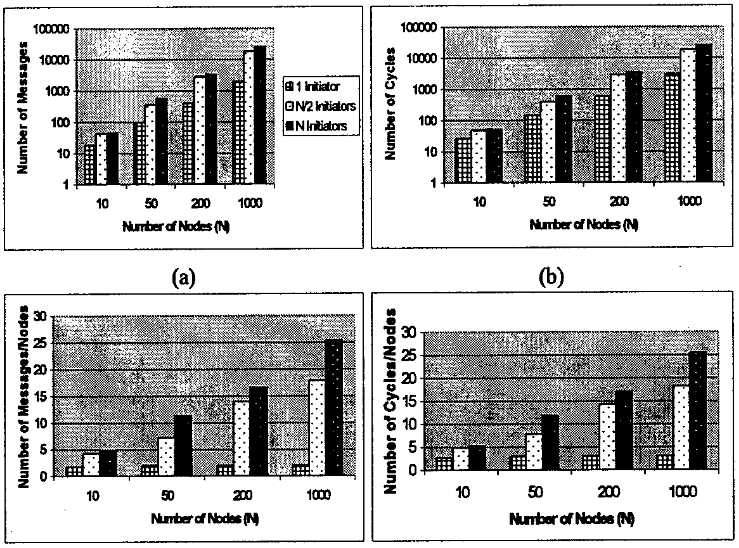
Figure 6: a) the schematic view of a four-legged CONRO robot. b) The node organization for the four-legged CONRO robot. The robot has initiated two tasks.

The DISTINCT algorithm is also used on CONRO robot for many other behaviors. Enabled by this method, the CONRO robot does not need to have any fixed "brain" modules and has "an especially spectacular ability to adapt on the fly" (reported by SCIENCE, 8/8/2003 [101]). Specifically, the robot can perform online *bifurcation, unification*, and *behavior-shifting*. For example, a moving snake may be bifurcated into pieces, yet each individual piece continues behaving as an independent snake. Multiple snakes can be concatenated (for unification) while they are running and become a single but longer snake. For behavior-shifting, a tail/spine module in a snake can be disconnected and reconnected to the side of the body, and its behavior will automatically change to a leg (the reverse process is also true). In fault tolerance, if a multiple legged robot loses some legs, the robot can still walk on the remaining legs without changing the control program. All these abilities would not be possible if modules could not cope with the topological changes in the configuration network. The movies of these remarkable physical self-healing can be found at our website http://www.isi.edu/robots. In fact, one cannot appreciate the full significance of these accomplishments without seeing these movies.

We have also evaluated the performance of the DISTINCT algorithm in the simulated SRSs consisting of N = 10, 50, 200, and 1000 agents. Each agent has four connectors for connecting to other agents. Configuration of the networks is randomly generated and for each configuration we randomly selected a subset of 1, N/2 randomly selected and N agents to initiate tasks. Each experiment is performed five times and averaged.

Figure 7a shows the number of messages sent by the agents. Figure 7b shows the total number of cycles required for solving each distributed task negotiation problem on a logarithmic scale. Cycles are the number of times that an agent executes a loop to check the received messages and send new messages. Figure 7c and Figure 7d show the average number of messages per node and the average number of cycles per node, respectively.

As we can see, when there is only one initiator in the network no conflict is generated. In these cases, each node needs only two messages for each child and one message for its parent to build a tree that links all nodes and therefore the average number of messages and cycles per node is constant. When multiple tasks are competing in the network the number of messages and cycles per nodes increases because more nodes must build and merge partial spanning trees and switch their tasks. The experiments show that in all cases, DISTINCT algorithm ensures that all nodes select one and only one task in a distributed manner and the cost is of the low polynomial order with respect to the number of competing tasks.



(a) (b)

(c) (d)

Figure 7: a) The number of cycles. b) The number of messages. c) The number of cycles/node. d) The number of messages/node.

To summarize this section, we presented a distributed algorithm called DISTINCT as a solution for distributed task negotiation in a self-reconfigurable system. The algorithm allows a large number of distributed agents to agree and select a task from many competing choices and terminate the negotiation synchronously. The algorithm is proved correct in acyclic graphs and it optimality was discussed. The time complexity of DISTINCT algorithm is of the low polynomial order respect to the number of competing tasks. We experimentally evaluated the DISTINCT algorithm in the domain of self-reconfigurable robotics and simulation. The results show that this algorithm is scalable and ensures that in all cases all nodes select one and only one task.

3.3.2 Distributed Behavior Selection

After all agents in the SRS agreed on the same task, they are left with the problem of accomplishing the selected task by generating a group behavior. A group behavior is the result of the coordinated performance of the individual agents' behaviors. In this section, we will describe how the behaviors are getting selected.

A SRS consists of a network of homogeneous agents. The only thing that differentiates the agents from one another is their location in the network relative to other agents. This is called the agent's *type*, (it will be described in detail in the next section). For example, a four-legged shape self-reconfigurable robot consists of the agents of types 'front right leg', 'front left leg', 'back right leg', 'front right leg', 'front spine', and 'back spine', See Figure 8a.

The types of the agents in a SRS can uniquely identify them in the network and therefore agents can use their type to select the right behavior for themselves. For example, to accomplish the 'Move forward' task, agents of types 'front left leg' and 'back right leg' will select 'Swing Backward' behavior, agents of types 'front right leg' and 'back left leg' will select the 'Lift up and Swing Forward' behavior and agents of types 'front spine' and 'back spine' will select 'bend left' and 'bend right', respectively. Figure 8b shows the robot after the agents have performed their selected behaviors.

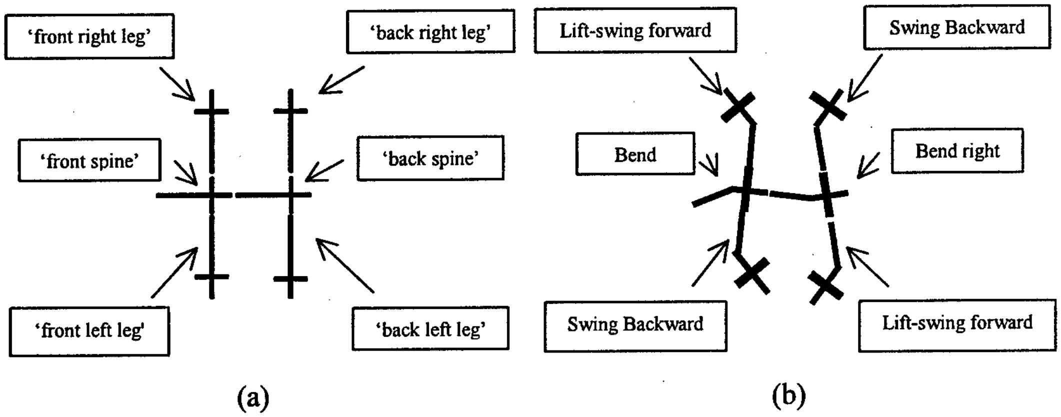
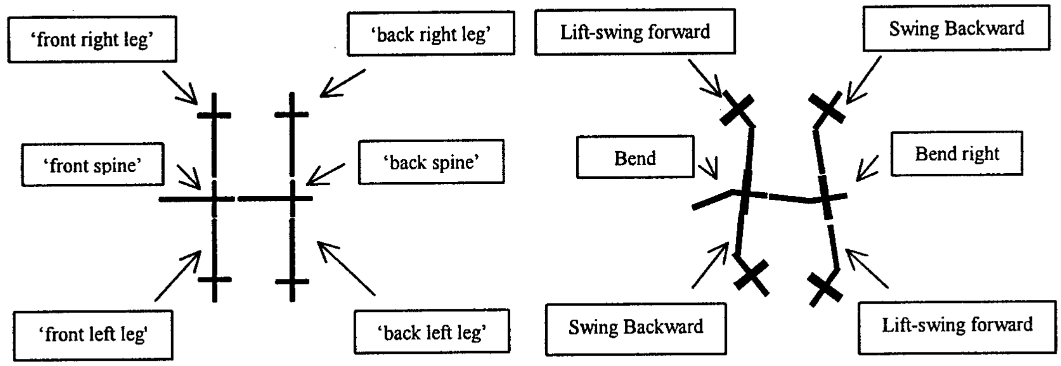


Figure 8: (a) The types of the agents in a four-legged self-reconfigurable robot. (b) The selected behaviors of each agent.

For example, if in the previous example 'Move Forward' was the selected task, a group behavior should be selected that moves the robot forward. A possible group behavior for accomplishing this task is shown in Figure 9. This group behavior is called the 'four-legged moving forward' and consists of the following behaviors: 'front left leg' and the 'back right leg' perform the 'Lift up and Swing Backward' behavior, while the 'front right leg' and the 'back left leg' to perform the 'Swing Forward' behavior and then they switch their behaviors.

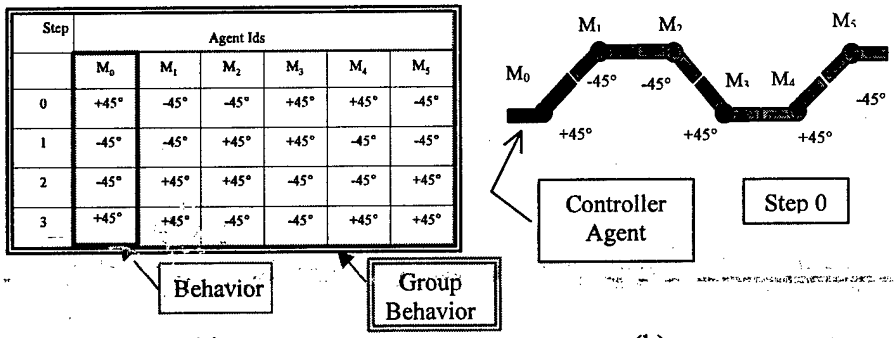
The 'Move forward' is a cyclic task (gait). Therefore, when agents complete performing their selected behaviors, they will continue with selecting another behavior. In the above example, agents of types 'front left leg' and 'back right leg' can now select 'Lift up and Swing Forward' behavior, agents of types 'front right leg' and 'back left leg' can select the 'Swing Backward' behavior and agents of types 'front spine' and 'back spine' will select 'bend right' and 'bend left', respectively. If agents continue selecting and performing these behaviors alternatively, a group behavior, called 'four-legged walk', will be generated that can move the robot forward and accomplish the task.


'front right leg'
'back right leg'
'front spine'
'back spine'
'front left leg'
'back left leg'
Lift-swing forward
Swing Backward
Bend
Bend right
Swing Backward
Lift-swing forward

(a) (b)

Figure 9: (a) Modules at different parts of a four-legged CONRO robot. (b) Modules after selecting and performing their behaviors.

So far, we saw that if agents in the network know their type in the network, they can use this information to select behaviors that can accomplish the selected task. Now the question is that how agents can find out what is their type? Previous approaches for solving this problem include approaches, where the human designer of the system specifies and assigns types to the modules (agents) in the network.



Step	Agent Ids					
	M_0	M_1	M_2	M_3	M_4	M_5
0	+45°	-45°	-45°	+45°	+45°	-45°
1	-45°	-45°	+45°	+45°	-45°	-45°
2	-45°	+45°	+45°	-45°	-45°	+45°
3	+45°	+45°	-45°	-45°	+45°	+45°

Behavior
Group Behavior
M_1
M_2
M_5
M_0
-45°
-45°
M_3
M_4
-45°
+45°
+45°
+45°
Controller Agent
Step 0

(a) (b)

Figure 10: a) A central 'gait control table' controller for a caterpillar shape robot. b) A caterpillar robot performing step 0. Controller agent is the left-most agent, M_0

The centralized controller system that uses a 'gait control table' [8] is an example of such approach. In this approach, the developer of the table specifies the type of each agent and the behaviors are assigned to them based on their unique Ids. In this approach behaviors are stored in the form of angle values columns of the table.

Figure 10a shows the 'gait control table'. Each column of the table is a sequence of actions that an agent executes over time. Therefore, it is equivalent to the behavior of that agent and the combination of all columns of the gait table represents a group behavior. Although, in this representation the action, behaviors and group behaviors are represented, but the agents do not know why they have chosen these behaviors. This information is available to the designer of the table in form of the agents' type but is not explicitly represented in the table. This is the reason why the agents (or the central controller agent) are unable to autonomously select new behaviors to accomplish the task when the topology of the network changes.

Figure 10b shows a caterpillar shape network consisting of six robotic agents performing the 'caterpillar move' group behavior (gait). The agents are performing the actions at the 'row 0' of the table. The Controller agent is on the left. The controller agent maintains the 'gait control table' and a counter called *step*, specifying the current row of the table. At each step, the controller agent sends the pre-specified actions on the current row of the table to the corresponding agents. After executing their actions, all agents send feedback messages to the controller agent and after it receives feedback messages from all the agents, the controller agent increment the *step* counter and sends the next row of the table.

In other approaches, agents are given explicit knowledge about their type in the network. Therefore, as oppose to the approaches based on the pre assigned types, these approach can dynamically adapt with the topology changes in the Self-Reconfigurable Systems and agents can autonomously select behaviors. Examples of such approaches are those proposed by [33] and [20]. In these approaches, agents use their local information about how they are connected to their neighboring agents, in order to detect their *local types* (types based on the local information). The local type of the agent is then used for selecting behaviors. For example, to generate a 'four-legged walk' group behavior in a four-legged robot, Figure , [33] and [20] identifies the agents with only one active connection link as agent of local type 'leg' agents (the four agents on the sides of the robot) and agents with more than one active connection links as agents of local type 'spine' agents.

Behavior selection based on the pure local information (local type), although very effective, is not enough for generating all possible group behaviors in a SRS. The following example shows a situation where behavior selection based on the local type cannot generate the desired group behavior. In this example, the goal is to generate two group behaviors called the 'caterpillar move' for the caterpillar shape self-reconfigurable robot shown in Figure 11a, and the 'butterfly move' for the T shape self-reconfigurable robot shown in Figure 11b. The 'caterpillar move' group behavior consists of a set of synchronized sinusoidal behaviors and the 'butterfly move' consists of the following behaviors: agents C and D move up and to the left then move down and to the right and agents A and B stand still. This moves the T shape robot to the left.

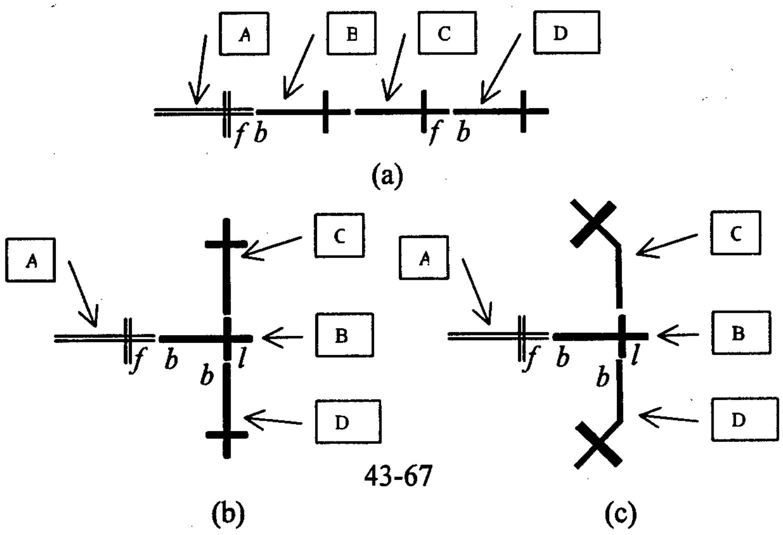


Figure 11: (a) a network of agents in the form of a caterpillar robot. (b) The same number of agents in the form of a T shape robot.

If we use the local connection information for selecting the behaviors to generate these two group behaviors, it can be seen that how the agents' B, C, and D connection links are connected to their neighboring agents (their local type) are different in these two robots; i.e. in caterpillar robot, the agent D's 'b' connection link is connected to 'f' connection link of its neighboring agent. While in the T shape robot, the agent D's 'b' connection link is connected to the 'l' connection link of its neighboring agent. Therefore agents B, C, and D can use their local type to correctly select a behavior based on the shape of the robot. However, this is not the case for the agent A. In both robots, agent A has the same local connection links. This is while agent A is expected to perform a sinusoidal behavior in the caterpillar robot configuration and a no movement behavior in the T shape configuration. This shows that specifying the type of the agents based on the pure local information is not enough to generate all possible group behaviors.

Table 1 shows all 32 possible type(0) of a CONRO module. For example, according to this table, the type(0) of agent A and B in are T2 and T21 respectively.

Table 1: All possible local types of a CONRO module.

	This Module				
	b	f	r	l	Type
Connected to neighboring modules					T0
	f				T1
		b			T2
			b		T3
				b	T4
	l				T5
	r				T6
		b	b		T7
			b	b	T8
		b		b	T9
	l	b			T10
	l		b		T11
	l			b	T12
	r	b			T13
	r		b		T14
	r			b	T15

This Module				
b	f	r	l	Type
f	b			T16
f		b		T17
f			b	T18
	b	b	b	T19
f	b	b		T20
f		b	b	T21
f	b		b	T22
l	b	b		T23
l		b	b	T24
l	b		b	T25
r	b	b		T26
r		b	b	T27
r	b		b	T28
f	b	b	b	T29
l	b	b	b	T30
r	b	b	b	T31

Extended Types

In this section, we will introduce a new approach for specifying agents' types in a SRS. The types of the agents in this approach are called *extended types* (to be distinct from the local types). Extended types are an extension to the agents' local types. The extended types can solve the limitations of behavior selection based on the pure local information (local types) and can generate any group behaviors in a SRS.

The extended type, type(*n*), read as 'the type of order *n*', of an agent in a SRS is defined as how the active connection links of the agent are connected to the connection links of the agents of distance *n*. The distance between two agents is defined as the number of agents between the two agents. For example in Figure 12, the distance between agents A and C is one and the distance between A and B, which are immediate neighboring agents, is zero. The type(0) of agent A, will be (*bf*). Because, agent B is the only agent of distance zero from the agent A and the *b* connection link of agent B is connected to the *f* connection link of agent A. Extended type is written starting from the other agent. Therefore, the type(0) of agent A will be (*bf*) and not (*fb*). Type(1) of agent A, which shows how the active connection links of agent A are connected to agents of distance 1 will be [(*br,bf*),(*bl,bf*)]. In Figure 12, the type of order zero, type(0), and one, type(1), of all the agents are shown. Since agent B does not have any neighbor of distance 1, the type(1) for this agent is not defined.

It can be seen that the definition of agent local type or role given by [33] and [20] is equivalent to Type(0). In addition, global representation of the entire network for each agent is equivalent to [type(0),type(1), . . ., type(*d*)], where *d* is the largest distance in the network.

In the previous example of selecting a behavior based on the local type, although the Type(0) of agent A in both robots in Figure 12 were the same, [(*bf*)], but the type(1) of agent A in these two robots are different; i.e. the type(1) of the agent A in caterpillar robot is [(*bf,bf*)] and in the T shape robot is [(*br,bf*),(*bl,bf*)]. Therefore type(1) of the agent A can be used to select a required behavior for the agent A in these two robots. Using the extended types, agents can uniquely identify themselves in a SRS relative to other agents and select the appropriate behavior based on the given task. Now we need to answer this question that 'how the agents can detect their extended types in the network?' In the next two subsections we present solutions for this problem.

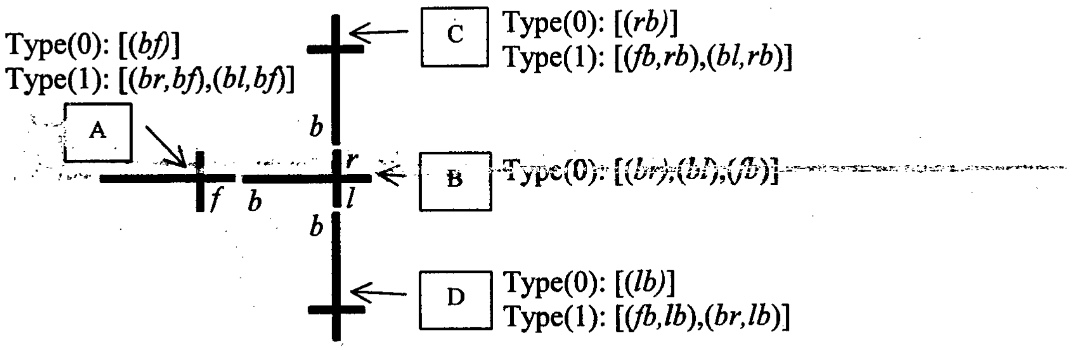

Type(0): [(bf)]
Type(1): [(br,bf),(bl,bf)]
A
f
b
b
r
l
b
C
Type(0): [(rb)]
Type(1): [(fb,rb),(bl,rb)]
B
Type(0): [(br),(bl),(fb)]
D
Type(0): [(lb)]
Type(1): [(fb,lb),(br,lb)]

Figure 12: Agent type and distance

Extended Type Detection Using Path

In this section we will introduce a distributed approach for determining the extended type of an agent in a SRS using the path field of the communicated messages. As we described before, the path field of a message consists of a sequence of the agents' connection links labels that is constructed as the message gets propagated in the network. For the agent D in Figure, we will see that the received path is a part of the agent D type(1) received from the agent A. Agent D can build its extended type(1) after it receives a messages from the agent C and appends the message path to its extended type. The items that are separated by comma in the path field of a message are called *path elements*. The number of the path elements, P, and the distance between the initiator of a message from the receiver of the message, d have the following relationships:

$$d = P - 1$$

Based in this relationship, the receiver of a message can distinguish which type order the received path belongs to and there it can build types of different order. For example, agent D can distinguish a message that is originated from agent C belongs to type(1) because the number of path elements in the path field of the message is two.

Behavior Selection Using Extended Types

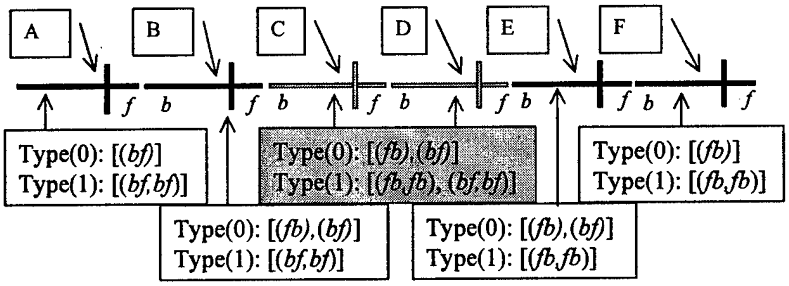
According to what was mentioned in the above, agents can detect their extended types based on the following algorithm: After the task selection phase is complete, initially each agent selects a behavior based on the information the agent has about its local active connection links with its neighboring agents (This information is the agent's type(0) and is available locally) and looking up in the 'Extended type to Behavior Mapping Table' (OTBMT), mapping an extended type (in this case type(0)) to a behavior. If no entry for the current local type was found, the agent chooses a null behavior. Then each agent sends a message to its neighboring agents. This message will be propagated to the other agents, while its path field is being updated. Meanwhile, agents update their extended types according to the path filed of the received messages. When an agent finds a match between a created extended type based on the received messages and an extended type in the OTBMT it selects the new behavior in the table. Eventually, all agents will detect their extended type and will be able to select to right behavior.

Although this approach can dynamically adapt to the changes in the topology of the network, it has two problems. First, an initiated message form an agent will be propagated to the rest of the agents in the network. This means that if there is N agents in the network the total number of communicated messages will be of order $O(N^2)$ (Since $N * (N-1) = N^2 - N$ messages will be communicated). This will be a problem in the networks with the large number of agents. The second problem is that in the network where the maximum distance of the agents is large, as the messages get propagated in the network, the size of the path field of the messages gets larger and as a result the size of the communicated message gets larger and larger. These two problems create bottleneck in the communication system in the large networks and slow down the response time of the network.

To solve these two problems, we set a maximum length for the path field of the communicated messages. For example, if the maximum length of the path filed is set to k, after there are k path elements in the path filed of the message, it will not be propagated to the other neighbors. In this situation, if there are N agents in the network, and each agent has the average number of a active connection links, the number of communicated messages will be of the order of $O(N)$ (since at most $a*N$ messages will be initiated and each message will be communicated k times therefore $k*a*N$ messages). In addition, the size of the path field of the message will not be larger than a maximum size, k, which will limit the size of the messages and also solves the second problem. However, this might cause that some agents not to be able to identify their extended type and as a result not to select the right behavior. In the next section we will solve this problem by creating a set constraints among the agents' selected behaviors.

Behavior Selection by Defining Constraints Among Behaviors

As we mentioned above, reducing the number of communicated messages, which is done to improve the responsiveness of the SRS causes another problem i.e. some agents will not to be able to fully detect their type and select a behavior. In order to solve this problem, we define a set of constraints among agents' selected behaviors. These constraints are in the form of selection rule shown in .



Rules are of the form
if (received *path* is X)
and (the *behavior* is Y)
then (select behavior Z)

If path = (*bf,bf*) **and** behavior = CAT_0
then select CAT_60

If path = (*bf*) **and** behavior = CAT_0
then select CAT_30

If path = (*bf,bf*) **and** behavior = CAT_30
then select CAT_90

If path = (*bf*) **and** behavior = CAT_30
then select CAT_60

If path = (*bf,bf*) **and** behavior = CAT_60
then select CAT_120

If path = (*bf*) **and** behavior = CAT_60
then select CAT_90

If path = (*bf,bf*) **and** behavior = CAT_90
then select CAT_150

If path = (*bf*) **and** behavior = CAT_90
then select CAT_120

If path = (*bf,bf*) **and** behavior = CAT_120
then select CAT_0

If path = (*bf*) **and** behavior = CAT_120
then select CAT_150

If behavior <> CAT_* **then** select CAT_0

Figure 1: The behavior selection rules

To summarize this section on distributed behavior selection, we have described how agents agree on the same task in a situation where multiple agents initiate many tasks and also after a task is selected, how to select behaviors that generate a group behaviors. What we did not address in the section was that agents couldn't detect when the task selection phase or behavior selection phase is done and the only thing they do in these situation is to wait indefinitely for new messages to arrive. Therefore, a separate mechanism is required to inform the agents about the termination of these processes. This mechanism is called the *synchronization mechanism* and the details of that can be found in our publication [1].

3.4 2003-2004 Period

The topology of a self-reconfigurable Robot can change anytime. This can be as a result of the failure of some modules of the robot, joining a new module to the robot, displacement of some module from one location to another as a result of the self-reconfiguration task or any combination of these cases. Considering that modules select their relevant behaviors to accomplish a given task based on the current topology of the self-reconfigurable robot, modules must be aware of the current topology of the robot and detect any changes to the topology. When changes to the topology of the robot are detected, modules can investigate new ways of accomplishing the given task. This document reports a distributed and dynamic way to discover topology and alter system function according to the discovered topology. We will describe the hardware and software architecture and algorithms for this capability and reports the demonstration we did for the CONRO self-reconfigurable robots.

Another major technical accomplishment in 2004 is the design of a distributed functional language called DH2 for programming of self-reconfigurable systems. DH2 is inspired by the biological concept of hormones and neurotransmitters. It provides messaging and execution constructs for distributed coordination among a collection of reconfigurable modules in accomplishing of global tasks. Through its constructs DH2 facilitates easy programming of locomotion and reconfiguration behaviors. DH2 is implemented as a meta-language on top of C++ and is tested on a set of simulated CONRO modules. Experimental results support the usability of DH2 for developing of autonomous self-reconfigurable systems.

3.4.1 Autonomous Discovery and Response to Unexpected Topology Changes

A self-reconfigurable system is a special type of complex systems that can autonomously or manually rearrange its software and hardware components and adapt its configuration (such as shape, size, formation, structure, or organization) to accomplish difficult missions in dynamic, uncertain, and unanticipated environments. A self-reconfigurable system is typically made from a network of homogeneous or heterogeneous *reconfigurable modules* (or *agents*) that can autonomously change their physical or logical connections and rearrange their configurations. Self-reconfigurable robots [1-4] are examples of such systems that consist of many autonomous modules that have sensors, actuators, and computational resources. These modules are physically connected to each other in the form of a configuration network. Since the topology of the network may change from time to time, the controller of the robot must be distributed and decentralized to avoid single-point failures, communication bottleneck among modules and to accomplish the given task.

These modules must have some essential capabilities in order to accomplish complex tasks in dynamic and uncertain environments. The capabilities that we addressed in our previous work were: (1) distributed task negotiation [5] – allowing modules to agree on a global task to accomplish, (2) distributed behavior collaboration [6] – allowing modules to "translate" a global task into local behaviors of modules; (3) synchronization – allowing modules to perform local behaviors in a coordinated and timely fashion; In these previous works we assumed the network of modules can have any initial topology but it remains unchanged during accomplishing a task.

Here we relax this assumption and allow the topology of the network of modules to change at any time including the middle of accomplishing a task. Our solution is a distributed approach inspired by the concept of hormones [10] and is based on 1) giving the ability of detecting local changes in the topology of the network to the modules and 2) letting them coordinate their activities in a new way such that the given global task is accomplished.

The related approaches for solving similar problems include Role-based Control [7] and stochastic approaches for self-repair such as [8]. The first approach is based on the changes in local relationships of the immediate neighboring modules. This approach requires less computational power. However, it is an open-loop approach and might not be very flexible for accomplishing complex tasks. The second approach has been applied to lattice-based self-reconfigurable robot which their configuration space is much smaller than that of the chain-type self-reconfigurable robots such as CONRO.

The problem of autonomous discovery and functional response to topology changes can be defined as follows: Given a global task and a set of self-reconfigurable modules, coordinating global responses to local changes in the topology of the network of modules in order to produce the desired global effects. Local changes include adding or deleting new modules or communication links to/from the network of modules.

This problem is very challenging due to several reasons: relationships among modules may change anytime; changes in configuration is locally detectable but a coordinated global response is required; the number of modules in the robot is not known; modules have no unique global identifiers or addresses; modules do not know the global configuration in advance, and can only communicate with immediate neighbors.

Generally, accomplishing a given global task is dependent on the topology of the network of modules [6]. Modules can accomplish a global task by selecting correct *behaviors* in coordination with other modules and performing them synchronously. As a result, changes in the topology will directly influences the behaviors that should be selected and the time they should be performed.

Formally, an autonomous discovery and functional response to topology changes problem is a tuple $[G(P, C), Q, T]$, where P is a list of nodes, p_i; C is a list of labeled physical or logical links, c_j, such that $j \in \{locally\ unique\ labels\}$; Q is the list of the internal state, q_i, associated with each node p_i, such that $i \in \{1, \ldots, N\}$; G is the *configuration graph* of the network of modules consisting of P nodes and C edges; T is the global task given to all nodes.

In cases where graph $g \in G$ can accomplish task T, and assuming that $Q \to T$ means the internal states of the nodes, Q, produce the desired behaviors to accomplish task T, an autonomous discovery and functional response to topology changes problem is solved if and only if $Q_g \to T$ meaning that the internal states of the nodes are a function of the topology of the graph such that the given task T is accomplished.

Here, the nodes and links represent the modules and the communication links between them, respectively. Note that the size of the network is dynamic and unknown to the individual nodes; also the index numbers are only used for defining the problem and not used in the solution.

Under these circumstances, a satisfactory solution to this problem must be distributed. Modules must detect local changes in the configuration graph and inform the rest of the modules in order to let them change their internal states.

To illustrate the problem, we use the CONRO self-reconfigurable robot as an example. CONRO is a chain-type self-reconfigurable robot developed at USC/ISI (http://www.isi.edu/robots).

Figure 1 shows the schematic views of CONRO module and a six-legged CONRO robot. Each CONRO module is autonomous and contains two batteries, one STAMP II-SX micro-controller, two servomotors, and four docking connectors for connecting with other modules. Each connector has a pair of infrared transmitters/receivers, called *outgoing-Links* and *incoming-Links*, to support communication as well as docking guidance.

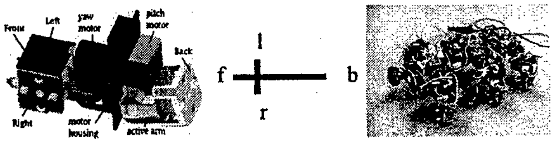


Figure 1: A CONRO module, the schematic view of one module. and a hexapod (insect) configuration

Each module has a set of open I/O ports so that various sensors for tilt, touch, acceleration, and miniature vision, can be installed dynamically. Each module has two Degrees Of Freedom: DOF1 for pitch (about 0-130° up and down) and DOF2 for yaw (about 0-130° left and right). The range of yaw and pitch of a module is divided to 255 steps. The internal state of each module includes the current values of the yaw, pitch of a module, and the number of the sent and received messages. The modules' *actions* consist of moving the two degrees of freedom to one of the 255 positions, attaching to or detaching from other modules, or sending messages to the communication links through the IR senders.

Modules can be connected together by their docking connectors. Connected docking connectors are called *active* connectors. Docking connectors, located at either end of each module. At one end, labeled *back* (*b* for short), there is a female connector, consisting of two holes for accepting another module's docking pins. At the other end, three male connectors of two pins each are located on three sides of the module, labeled *left* (*l*), *right* (*r*) and *front* (*f*).

Probing and Communication

The first step for the modules in responding to the network topology change consists of detecting local changes. Instances of local changes are: 1) When a new module connects to one of the modules in the network, 2) When an existing module disconnects from all other modules in the network, 3) When an existing module establishes a new connection with another module in the network, and 4) When a module disconnects some of its connectors from other modules in the network. In situations 1 and 2 the number of nodes and in situations 3 and 4 the number of nodes and links in the configuration network changes.

Modules can detect local changes in the topology of the network by periodically monitoring their active docking connectors for disconnections and inactive docking connectors for new connections. This action will be called *probing*. In order to detect all the above-mentioned cases of topology change efficiently, we will use two types of probing: 1) Probing when modules are communicating and 2) Probing using *probing signals*.

The communication protocols between modules that use handshaking signals when sending and/or receiving messages can be used for probing the active connection links between modules. A successful communication action over an active connector shows that the connection is still active. Similarly, an unsuccessful communication action shows the disconnection of an already active connector.

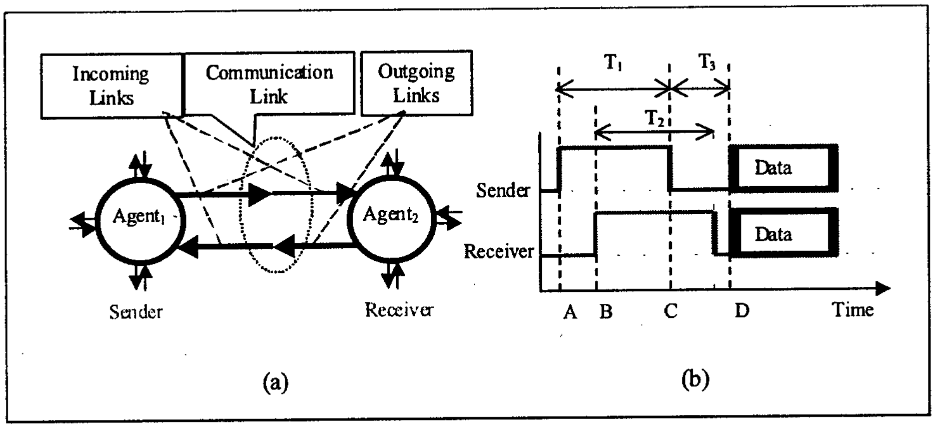


Figure 2: (a) The communication link and (b) the asynchronous communication protocol between two agents

Figure 2 shows an asynchronous communication protocol that was implemented in CONRO modules. Agent$_1$ is the sender and agent$_2$ is the receiver. What follows is a brief description of the handshaking sequence of this protocol:

1) The sender requests to send a message by making its outgoing link 'High', point A. and then frequently checks its incoming link for receiving a 'High' signal.
2) The receiver responds by making its outgoing link 'High', point B, and waits.
3) After receiving the 'High', sender makes its outgoing link 'Low', point C, and starts sending the message (Data) after some delay, point D. This short delay is called *preparation time* (T$_3$), which gives a chance to the receiver to prepare for receiving Data. Data is communicated using RS232 asynchronous communication protocol. T$_1$ and T$_2$ are the timeouts of the sender and receiver, respectively.

This simple handshaking protocol successfully completes if and only if both modules actively participate. Therefore a successful communication verifies an active link between two modules. Oppositely, an unsuccessful communication confirms the receiving module is not present and the link is inactive.

This method of probing, however, is not an efficient way of probing the inactive docking connectors unless; an attempt to send a message and waiting for the timeout is what we have been looking for. Also, in situations where two modules do not communicate for periods that are longer than monitoring period, the communication-based approaches will not be useful. In such situations we will use a different probing method based on sending *probing signals*.

Probing Signals

Probing signal are narrow pulses that are periodically sent to inactive connections or active connections if no communication occurs on them for a long time. The width of probing signals is quite narrow such that they can be distinguished and filtered from the communication protocol signals. Figure 3 compares the probing and the communication protocol signals widths.

Figure 4 shows the block diagram of a module's connection link. The 'Probing Signal Filter' on the incoming link separates the probing signals from the communication signals. On the outgoing link, the communication and probing signals are merged on a single output line.

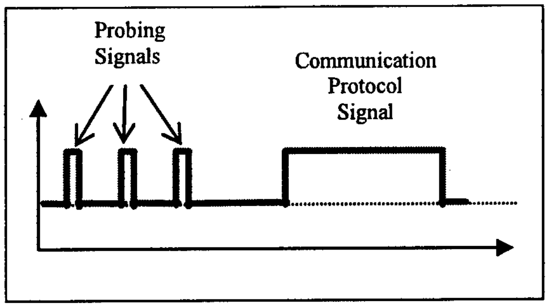


Figure 3: Probing and Communication protocol signals;

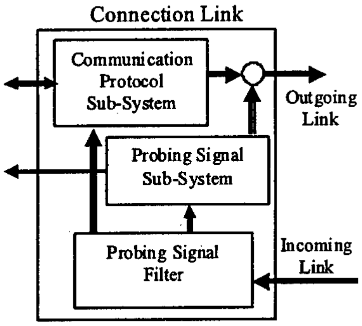


Figure 4: Joining and separation of the probing and communication signals

```
when GenerateProbe () do
  for each C ∈ Connectors do
    if (C = Inactive) or (NoComm (C, Period) = true) do
      send ProbingSignal to C;
 end do; end do; end do;

when CheckTopology () do
  TempLocalTopology = CurrentLocalTopology;
  TopologyChanged = false;
    for each C ∈ Connectors do      //reset
          CurrentLocalTopology (C) = Inactive; end do;
  for each C ∈ Connectors do
        if (CommOccurred (C) = true) or
           (Probe Signal Received (C) = true)
do
          CurrentLocalTopology (C) = active;
        end do; end do;
  if (TempLocalTopology ≠ CurrentLocalTopology) do
        TopologyChanged = true;
  end do;
  return TopologyChanged;
end do;
```

Figure 5: The Probing Algorithm

Probing Algorithm

Figure 5 describes the probing algorithm for detecting local changes in the topology of the network. This algorithm consists of two procedures. The first procedure, *GenerateProbe*, is called for generating probing signals on the inactive connectors or the active connectors that have not communicated for longer than 'monitoring period'.

The second procedure, *CheckTopology*, is called for detecting changes in local topology of the network based on the received probing signals or the recent communicated messages. This procedure returns a true value if the topology has been changed.

Functional Response to Unexpected Topology Change using probing

Our solution for the functional response to topology change problem in self-reconfigurable robots relies on our previous work on distributed control for self-reconfigurable robots. Specifically, the 'distributed task negotiation' and 'distributed behavior collaboration' problems. In this section we will briefly describe these problems and their proposed solutions. Then we will present our algorithm that is based on probing for solving the functional response to the topology change problem.

Distributed Task Negotiation is a process by which modules in a self-reconfigurable robot can negotiate and select a single coherent task among many different and even conflicting choices.

In [9] we presented the DISTINCT algorithm as a solution for the distributed task negotiation problem. The main idea is that all modules work together to build global spanning trees and each tree is associated with a task. Initially, all modules that have their own competing tasks start building their own trees, but as they exchange messages for tree building, most modules will give up their "root" status and participate in building trees for other tasks. In this process, modules report their status to their parent module in the tree that they participate, and the module that does not have parent but received reports from all its children is the root for the entire network of modules. When this happens, this root module can conclude that the negotiation process has succeeded and all modules in the tree have agreed on the same task. An embedded synchronization algorithm detected the termination of the negotiation process.

Important characteristics of this solution are: 1) modules do not require having unique Ids; 2) ensures that all nodes will select the same task coherently; regardless of the number of competing tasks initiated in the network; and more importantly 3) it is not dependent on the topology of the network of modules.

Distributed Behavior Collaboration is a problem defined as follows: Given a global task and a group behavior, selecting a correct set of local behaviors at each module and coordinate the selected behaviors to produce the desired global effects.

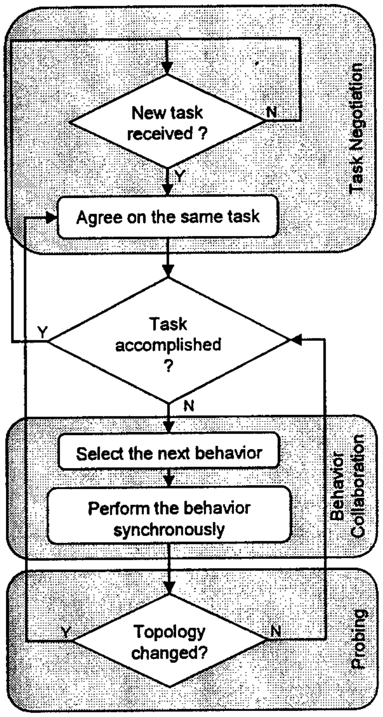


Figure 6: The FEATURE algorithm

In [6] we presented D-BEST algorithm. It is a new approach to distributed behavior collaboration based on the concept of "path" to represent extended neighborhood topology at the connector level. This allows modules to select appropriate local behaviors for a given global task in a given configuration, based on the location of the modules relative to other modules. D-BEST algorithm utilizes the modules' *extended type* for behavior selection. The number of communicated message were reduced from $O(N^2)$ to $O(N)$ (where N is the number of modules) by introducing a set *decision rules* representing the relevant behaviors for accomplishing a given task. The same embedded synchronization algorithm used for the task negotiation algorithm was used here for synchronized execution of the modules behaviors.

The FEATURE Algorithm

In this section, we will describe the FEATURE algorithm that brings all the above-mentioned pieces together and solves the problem of functional response to topology change in self-reconfigurable robots. This will be the algorithm that will run on all modules of the self-reconfigurable robot to ensure the homogeneity of all modules. Figure 6 depicts this algorithm.

Initially all modules will wait to receive a new task. The new task can be initiated by an outside controller or by one of the sensors of a module. Receiving a new task initiates a distributed negotiation process among all modules in the robot. This is necessary to ensure that 1) all modules know what task they accomplishing and 2) in cases where multiple modules have initiated more than one task, all modules will agree on accomplishing the same task that has the highest priority. This process is controlled by the DISTINCT algorithm for task negotiation shown on top of the Figure 6.

If the selected task is not already accomplished, modules will generate a set of relevant behaviors. The relevant behaviors are represented by a set of decision rules that have been downloaded in all modules. The execution of the selected behaviors will be coordinated by an embedded distributed synchronization mechanism. The above-mentioned process is controlled by the D-BEST, behavior collaboration algorithm shown in the middle of the Figure 6.

If the topology of the network of module changes while modules are performing their behaviors, the modules that detected the local changes, will initiate the DISTINCT algorithm using the current selected task in order to dynamically create a new spanning tree for synchronizing and initiating new sets of behaviors based on the current topology of the network. The topology detection process will be controlled by the probing algorithm shown on bottom of the Figure 6.

Experimental Results
We have implemented and tested the FEATURE algorithm and all of it sub–algorithms on the CONRO self-reconfigurable robots. All modules are loaded with the same control program and decision rules. For economic reasons, the power of the modules is supplied independently through cables from an off-board power supplier. But all modules are running as autonomous systems without any off-line computational resources.

In our experiment we gave a 'Move' task to a quadruped CONRO robot. The robot initiated a 'Four-Legged Walking' gait. While performing the gait, we detached the two spine modules. The resulting configuration was two separate T-shape robots. In this situation each T-shape robot continued the locomotion by executing the 'Butterfly Stroke' gait. Later, two T-shape robots were re-connected and the resulting four-legged robot re-initiated the 'Four-Legged Walking' gait. The videos of these experiments are available at (http://www.isi.edu/robots).

3.4.2 DH2: Distributed Functional Language for Self-Reconfigurable Systems

A *Self-Reconfigurable System* is a special type of complex system that can autonomously rearrange its software and hardware components. Such system is able to adapt its configuration (such as shape, size or formation) to accomplish difficult missions in dynamic and uncertain environments. A *Distributed Self-Reconfigurable System* is a network of autonomous, homogeneous or heterogeneous reconfigurable *modules* (*agents*). Nodes of this network have a special ability to change their physical or logical connectivity but also can perform generic, task-oriented actions. Distributed homogeneous systems have the advantage of being robust – since they have no singe point of failure, are easy to repair and can be cheap to produce because of modular equivalence. The rest of our discussion will focus on systems of this type.

A *Self-Reconfigurable Robot* is an embodied self-reconfigurable system that has a defined morphology, computational facilities and a set of actuators, sensors and docks. Such robot offers many potential advantages over robots of a conventional design, with the main advantage being multi-functionality. For example a self-reconfigurable robot could become a "snake" to slither into tight spaces that are hard to reach by humans or conventional robots. Then, it could morph into an "octopus" and transform a leg into a gripper to manipulate objects. Such a robot could divide itself into multiple independent agile units to accomplish tasks that require simultaneous actions in different locations. A single self-reconfigurable robot could perform transportation, inspection, assembly and many other functions with potentially much less cost than a large collection of specialized conventional robots. Some steps have been made to realize this vision [11-13], however a lot of problems remain to be solved before it is accomplished.

The challenges in controlling of self-reconfigurable systems are abundant. Such a system has two levels of behaviors that are highly coupled. On the low level local actions of each module are determined by its state, sensor data and communication with other modules. On the high level the whole system interacts with the environment to accomplish its current task. Part of the high level behavior can be switching to a different configuration, which in its turn can change behaviors of some modules. This circular interaction between the two behavior levels makes controlling of self-reconfigurable systems a complex problem. For the system to be functional, each module has to dynamically specialize its actions based on its position within the system so that local actions do not conflict but complement each other in accomplishing the current task. To act purposefully, modules have to be aware of the system's global state, environment conditions and the status of the task completion. Thus, input data from heterogeneous or homogeneous sensors must be integrated and redistributed among all the modules. On the high level, a self-reconfigurable system must decide not only how to use its current configuration, but also what configuration will be the best for the given task and environment, and how to switch to that configuration with least cost. All these challenges make conventional centralized control methods unsuitable for reconfigurable robotic systems. Generic and distributed control architecture is required to make design and programming of such systems feasible and efficient. We have identified the following features such architecture has to support:

- *Decentralized Control:* The architecture has to be distributed to avoid having a single point of failure. Thus, global "names", "identifiers" or "addresses" should not be used.
- *Role-Based Functionality:* The architecture must enable modules to function according to where they are (i.e. what *role* they play) in the current configuration. The *role* should be dynamically assumed by each module based on its local state, topological type and information received from other modules.
- *Dynamic Cooperation:* The architecture has to support the ability of modules to cooperate in achieving a desired global behavior. No predefined leaders should be assumed, and division of labor among modules should be done through negotiation.
- *Topology Discovery:* Since new configurations are to be dynamically formed, the framework must provide modules with the ability to discover the new topology of the system through communication with their neighbors. This is a critical capability for dynamic cooperation and self-healing.
- *Global Synchronization:* The architecture has to support synchronization without assuming any global o'clock. This is required since for most tasks local actions of modules must be synchronized to produce the desired global effects.
- *Scalability:* Since an optimal configuration for a given task is to be dynamically discovered during mission, the control architecture has to support any shape or configuration of arbitrary sizes.
- *Adaptability:* The architecture has to support ability of the system to evaluate the system's performance and reconfigure based on that evaluation.

Overview of DH2

To address the challenges in programming and controlling of self-reconfigurable robotic systems we propose a distributed programming language DH2. Our approach mimics the hormonal and neurotransmitter communication mechanisms existent in most biological species. A self-reconfigurable system can be considered as a graph with each node being an autonomous cell with its own power, processors, actuators, sensors, and connectors. A node can send "hormone" messages to communicate with other nodes. Sending of a hormone can be triggered by the already present hormones or by the environmental/task stimuli. Each node has a dynamic set of receptors for binding and processing of hormones. When a receptor binds to a hormone actions of the receptor are executed. Local actions include perception, locomotion, manipulation; as well as creating and destroying links with other modules, generating and propagating hormones. The first hormone in the system may come from an external source (e.g. a human operator), or created by an existing receptor when triggered by a sensor input.

To summarize, execution of the receptors' actions is triggered by hormones, and generation and consumption of hormones is performed by receptors (Fig 1). This style of execution is scalable with the system's shape and size because all modules are created with identical sets of receptors and do not need to know the entire configuration to be able to cooperate. The described mechanism is capable of producing distributed locomotion behaviors and distributed changing of the current system's configuration to a new one.

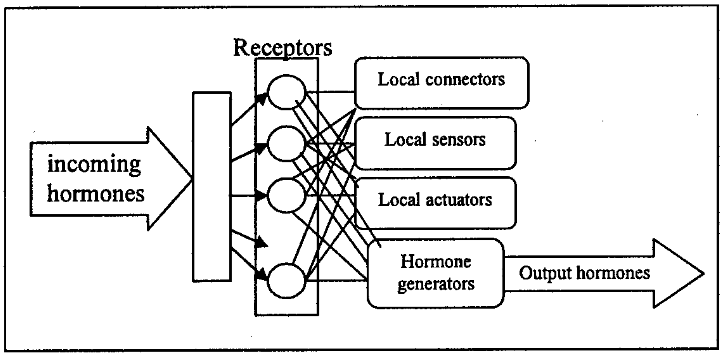


Figure 7: The structure of receptor-hormone control mechanism.

The concept of Hormones has been used before in computer science and robotics research. This includes Autonomous Decentralized Systems [14], homeostatic robot navigation [15] and integration of behaviors using hormones [16]. To our knowledge the concept of hormones was first applied to the problem of autonomous distributed reconfiguration and experimentally tested in our lab [17-19] The current work is an attempt to generalize and formalize the Hormone-based control method as a distributed programming language.

DH2 differs from other distributed programming languages in the way it implements messaging between system nodes. The hormone-like messages are similar, but not identical, to the content-based messages. They do not use global identifications to propagate through the network, which are commonly used in other distributed languages [20]. The hormone propagation is also different from generic message broadcasting because a hormone may be modified during its propagation and there is no guarantee that every module in the network will receive the same copy of the original message. Hormones are similar but not identical to the pheromones [21, 22] since hormones propagate from cell to cell without leaving residues in the environment.

The "Phase automata" programming model [23] presents an interesting way of generating locomotion gates for modular robotic systems. It alleviates the necessity for gate tables and allows hierarchical representation of complex behaviors distributed among groups of modules. Different from DH2, the model uses unique module ids and explicit mappings of behaviors to modules. The "Phase automata" model limits itself to periodic locomotion behaviors, whereas

DH2 supports behaviors of arbitrary temporal structure and behaviors that could involve computations, sensing and other available functionalities. Other related work includes distributed sensor networks [24], swarm robotics [25] and high-speed network protocols [26].

DH2 specifications

Any programming framework can be divided into two parts: a program itself and an execution environment. A program is a symbolic representation of the author's intent regarding what a computer or robot should do. An execution environment is a set of facilities that map the program's instructions onto computational or physical actions. Such an environment could be implemented in various forms: a compiler, an interpreter an external library, a virtual machine etc. In this section we try to describe DH2 in a generic way, abstracting from a specific environment implementation. In the next section we will present how this description could be implemented as a meta-language on top of C++.

A DH2 program consists of one or more *receptor* definitions (1). Receptor is defined (2) by its triggering *pattern* and a list of *actions*. Receptor R_i also has a numeric id and can be active or passive. Receptors are defined within the module's *state* which is created and updated by the execution environment. The *state* of a module is defined (3) by its *topological type* T and four sets of values: S – the values for each *sensor*, V – the values for each *local variable*, H – the presence or absence of each *hormone*, R – the activity *status* for each *receptor*.

<program> → <receptor> {, <receptor>} (1)

<receptor> → <pattern>, <action> {, <action>}, id, <status> (2)

<state> → <factor> = value {, <state>} (3)

<pattern> → <factor> (== | !=) value | <pattern> (and | or) <pattern> (4)

<action> → computation | actuation H_i = <status> | R_i = <status> (5)

<status> → 0 – passive/absent | 1 – active/present (6)

<factor> → T | S_i | V_i | H_i | R_i (7)

Figure 8: Extended BNF representation of the DH2 program

A triggering pattern (4) of a receptor defines a subset of module states that cause execution of the receptor's actions. The simplest pattern is represented by a statement of equality of one state's element to some value (e.g. H_i == 1 - hormone i is present; S_3 == 10 - sensor i reads 10). More complex patterns can be constructed recursively from simpler patterns using logical operators. An action (5) of a receptor is either an execution of some functionality available in the module or a specific change in the module's state. The latter includes binding to the present hormones or activation/inhibition of other receptors by changing the corresponding state variables.

Execution of a DH2 program proceeds in the infinite loop composed of two parts: receptor execution and "bookkeeping" (Fig 9). In the first part, for each active receptor its pattern is compared to the module's state and, if successfully matched, the receptor's actions are executed. In the second part the state of the module is updated through receiving of new hormones, topology discovery, connector status monitoring, readings sensor values etc. Since hormone propagation is a crucial part of the DH2 execution environment we will describe it in detail.

```
module_process (ClockCycle) →
loop () →
```

```
    for each active receptor Ri
        if match( <state>, Ri <pattern>)
            do all Ri <action>
            (place generated hormones in Buffer)
    for each hormone(_, [(x, y) | path]) in Mailbox
        do ConnectorStatus [y] = x,
    for each hormone (_, [(z, _) | path]) in Buffer
        do remove and send it via the connector z
    if send fails (e.g. no receiver at the connector z)
        do ConnectorStatus [z] = 0
    LocalTimer = mod (LocalTimer + 1, ClockCycle)
end
```

Figure 9: Main loop of the DH2 execution environment

A Hormone has the format (*content, path*), where *content* can be any term in DH2, and *path* is a list of connector names through which the hormone has been propagated. A newly created hormone has an empty path. If a hormone is propagated through a link l(x, y), then its path is assigned to [(x, y)| *path*]. If a hormone received by a module does not bind to any of its receptors, it will be forwarded to all the links of the module, except the last link in the hormone's path (i.e. its source). In the acyclic configurations this algorithm assures that a hormone generated by one module gets to all other modules without conflicts. Also the path of a hormone is used to generate "reply" hormones with the original source as their destination.

The execution environment maintains a Mailbox for received hormones and a Buffer for hormones to be propagated. It also monitors the status of each connector so that a module can dynamically adapt to changes in the network and discover its local topology in time. The local topology of a module is defined by a set of variables ConnectorStatus (e.g. {front=back, back=0} = T2 - tail). Initially all ConnectorStatus [*] = 0. If a module's connector x is in a link l(x, y), then ConnectorStatus[x] is set to y when the module receives a hormone through x. Since every module attempts to receive hormones from its connectors in every cycle of the program (Fig. 3), the ConnectorStatus will be updated correctly whenever there is a change in the local links.

Locomotion and reconfiguration programs

For the purpose of this section we will assume that DH2 is implemented according to the specifications and that the execution environment provides the following function to define a receptor:

$$\text{receptor}\ ([P_H(H), P_T(T), P_S(S), P_V(V)],\ [\text{action}_1(\text{args}_1), \ldots, \text{action}_J(\text{args}_J)]) \tag{8}$$

The first part of the function's expression specifies the pattern-matching functions P_H, P_T, P_S, P_V, and their parameters: H - the expected hormones, T - the expected local topology, S - the expected local sensor values, and V - the expected local variable values. The second part of the expression specifies actions to be executed by the receptor when all of the pattern-matching functions evaluate to true. Using this definition we proceed to an example of a control program for snake locomotion.

To make a CONRO snake configuration move in a caterpillar gate each module's pitch motor (DOF1) should go through a series of positions (e.g. +45°, -45°, -45°, +45°). However, these local actions should be synchronized in such a way that their global effect is a forward movement of the whole configuration. To coordinate the actions among modules a set of hormones and receptors will be used. Using hormones each module will inform its immediate neighbor what action it has selected so that the neighbor can make the appropriate choice itself and continue the hormone propagation.

TABLE I. DH2 PROGRAM FOR THE CATTERPILLAR GATE

Module type	Local Timer	H	DOF1	New hormone
T1	0		+45	[(X, A), b]
T1	(1/4)ClockCycle		-45	[(X, B), b]
T1	(1/2)ClockCycle		-45	[(X, C), b]
T1	(3/4)ClockCycle		+45	[(X, D), b]
T16, T2		(X,A)	-45	[(X, B), b]
T16, T2		(X, B)	-45	[(X, C), b]
T16, T2		(X,C)	+45	[(X, D), b]
T16, T2		(X,D)	+45	[(X A), b]

To implement this gait in DH2 we define 8 receptors and 4 hormones (Table I). Specifically, the first four receptors will cause the head module (T1) to generate new hormones and send them to the rest of the snake. The creation is triggered based on the value of the variable LocalTimer that results in four hormones per ClockCycle. The last four receptors will cause all the body modules (T16) to set their DOF1 to corresponding angles. These modules will receive hormones through the front connector f and propagate hormones through the back connector b. When a hormone reaches the tail module (T2), the propagation will stop because this module's back connector is not connected to any links. The speed of the caterpillar gait is determined by the value of the ClockCycle. The smaller the value is, the more frequent the head generates new hormones, and thus faster the caterpillar moves. The first and the fifth receptors for the caterpillar gate can be created using the provided function (8) as follows:

receptor ([= (T, 1), = (LocalTimer, 0)], [DOF1 (+45), hormone ([X, A), B]), keep_alive ()])

receptor ([= (T, {16, 2}), = (H, (x, A))], [DOF1 (-45), hormone ([X, B), B]), keep_alive ()]).

The above example works for snakes of arbitrary length and allows modules to be arranged randomly in the configuration. Similar advantages can also be achieved in controlling the process of reconfiguration. A single hormone will be sufficient to trigger a change of one configuration into another, and there will be no need to give low-level instructions to individual modules. For example, to reconfigure a legged CONRO robot into a snake, the action sequence could be as follows: the robot first connects its tail to a foot, and then disconnects the connected leg from the body so that the leg becomes a part of the tail. This compound action is repeated until all legs are "assimilated". The lower-level actions that implement this process are those that enable the tail to find a foot, to align and dock with the foot, and then disconnect the leg from the body [27].

This action sequence can be implemented using hormones and receptors as follows. The reconfiguration process is triggered by introducing an LTS (Legs To Snake) hormone into the system through any module. This hormone eventually propagates to all modules, but only the foot modules (T5 or T6) have the receptors to react to the LTS hormone and generate a "reply" hormone RCT (Request to Connect to the Tail). Every foot will periodically generate an RCT as long as it is still a foot. Only the current tail module (T2) has a receptor for RCT, and upon receiving an RCT, this receptor will acknowledge the sender (using the path in the received RCT) with a TAR hormone (Tail Accept Request), and will terminate itself so that no more RCT will be bound at this tail module. When receiving the TAR hormone, the selected foot module stops generating RCT hormones, and generates a new hormone ALT (Assimilate Leg into Tail) to inform all the modules in the path to perform the actions of bending, aligning, and docking the tail to the foot. When these actions are accomplished, the new tail module will create a new receptor for accepting other RCT hormones, and another leg assimilation sequence will start. This process will repeat until all legs are assimilated, and it is independent of how many legs are in the current configuration. Thus, if the reconfiguration sequence was stopped unexpectedly or prematurely, the process can resume itself correctly after the interruption is over. To implement this sequence in DH2, four types of receptors must be in place to bind and react to LTS, RCT, TAR, and ALT hormones. These receptors can be created as follows:

receptor ([= (T, {5, 6}), = (H, LTS)], [periodical_hormone ([RCT, _])])

receptor ([= (T, 2), = (H, (RCT, path))], [hormone(TAR, path^{-1})])

receptor ([= (T, {5, 6}), = (H, (TAR, path))], [stop_hormone(RCT), hormone(ALT, path^{-1})]),

receptor ([= (H, ALT)], [assist_ leg_assimilation()])

The first receptor allows a foot to react to LTS and create RCT. The second receptor allows a tail to react to RCT and reply with TAR. The third receptor allows a foot module to react to TAR and acknowledge it with an ALT. The fourth receptor will enable all modules in the spinal cord to assist in leg assimilation.

Experimental Results

In order to test the proposed architecture experimentally we have implemented the DH2 programming environment as a meta-language on top of C++. The triggering pattern of a receptor is represented by a regular expression containing a set of the required environment factors and their values. For example a pattern for some periodic action by a tail module can be represented as follows: "(.*,((T2|T19)|t0),.*){2}", where "T2" and "T19" represent the topological type "tail" and "t0" - timer equal to zero. This regular expression can be matched with a string that contains "t0", and "T2" or "T19" in any order (e.g. "t0,T19", "T2,t0" but not "T3,t0"). State of each module is represented as a multidimensional array of values for hormone presence, sensor values, receptor status etc. In order to match the current state against patterns of each receptor the state is converted to a string of the form: "<factor><value>, <factor><value>, ...". Conventions that are used are: "xy" means local variable x is equal to y, "Hi" means hormone i is present, "Ti" means current topological type is i etc. The resulting string is matched against the patterns using C++ regular expression routines. If matching is successful, then actions of the corresponding receptor are executed. The actions are implemented as static function pointers accessible to any module process. The action functions accept a module pointer as their only parameter to be able to access that module's state variables. The module's state is encapsulated in a class DHController. An instance of this class gets attached to a Module object (Fig. 10) and when executed runs the infinite control loop as defined in the DH2 specification by executing receptors, sending corresponding commands to the module's actuators, updating module's state, propagating hormones etc.

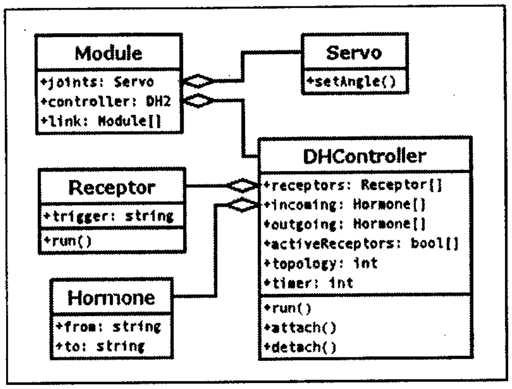


Figure 10: Class diagram of the DH2 environment

The implemented DH2 environment was used to control a set of simulated CONRO modules. Our goal was to create a reasonably close model of a CONRO robot in an environment that approximates the natural physics. After considering several commercial and free simulation environments, we decided to use Open Dynamics Engine (ODE) – an open source software library for simulation of rigid body dynamics, developed by Russell Smith. Using ODE's API we have developed a hierarchy of classes (Fig. 10) that represent static and dynamic properties of a virtual CONRO module.

Morphology of the CONRO model was intentionally simplified to make the simulation more efficient. A module is composed of three rectangular bodies joint by two actuators with pitch and yaw DOFs. Values for dimensions and masses of the bodies as well as maximum available force and angular speed for the servos were obtained from measurements and documentation. Relationship between dimension units of the simulation environment was set to represent the metric system. Since the physics engine has many free variables that determine stability and accuracy of the simulation, several prototype experiments were created to determine the optimal settings. Simulation of friction was set to a "pyramid model", with "contact slip" in both directions. Since the default setting for the friction direction produced inconsistent behaviors dependent on the model's orientation, the friction direction was dynamically updated in the module's frame of reference.

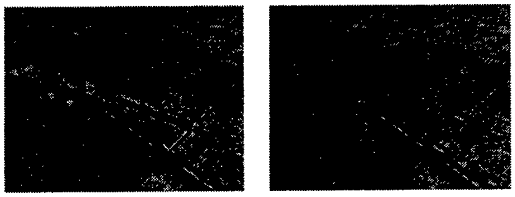
Locomotion in different configurations

Each experiment was conducted in the following manner: a set of modules and controllers were created, modules were positioned into the target configuration, the corresponding modules were docked to establish communication links between their controllers, and then the simulation was set to run. The following configurations were tested: snake of 1, 5, 8, and 20 modules, tripod, quadruped, hexapod, and octopod. For all of the above configurations, successful locomotion was generated by the same set of 20 receptors – four for each topology type: head, snake body, left leg, right leg, and spine. This demonstrates that the same DH2 program can be reused for different configurations.

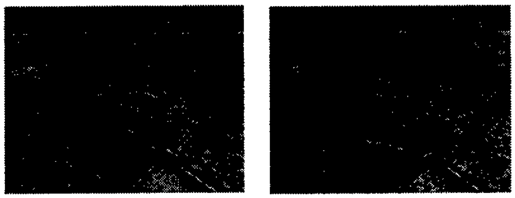
To demonstrate robustness of the control using DH2, we have done "live surgery" on a set of moving modules. A timed event was programmed to break one of the links and to split the original configuration into two separate ones. For example in one experiment an 8-module snake was broken into two, four and then eight separate snakes. At the moment of each division we observed snake body modules discovering their new topology types and taking over the leadership in the newly formed snake. In another experiment, a walking octopod was split into two quadrupeds that proceeded with their movement independently without any interruption or code modification. This demonstrates the utility of distributed control without using of unique identifications.

Reconfiguration

In the beginning of each experiment, modules were put into a certain configuration and left running in their locomotion gait for some time. Then a signal for reconfiguration was sent to an arbitrary module in the system, after which modules were supposed to change their configuration and continue locomotion using new gaits. The following morphology changes have been tested: an 8-module snake to a T-shape; a 9-module snake to a quadruped and a 3-module snake; a dragon to an 11-module snake. In addition to the locomotion receptors used in the first set of experiments, modules needed only a small set of additional receptors to successfully accomplish all of the above morphology changes. For example for the reconfiguration from a legged robot to a snake 10 additional receptors were used. Figure 11 shows some steps in morphing of a legged "dragon" to a snake configuration. Videos demonstrating this and other experiments can be found at our website http://www.isi.edu/robots/movies/.



Ready to reconfigure The tail has connected to a leg



One leg has assimilated into the tail The new tail connected to 3rd leg

Figure 11: Steps of the "dragon" to snake reconfiguration

Results of our experiments demonstrate that DH2 provides a flexible development framework for programming of distributed locomotion and reconfiguration behaviors. It is our belief that this framework can be successfully employed on other distributed platforms and it is one of our future goals to test that. In the current implementation, change of morphology is triggered by an external high-level signal. We plan to show in future that DH2 can be used to program autonomous discovery of the optimal configuration based on sensor data. If successful this could produce such desirable behaviors as adaptive locomotion and self-healing. Generalizing of the DH2 model to the level of an encoding scheme could make it possible to use Genetic Algorithms for generation of new behaviors and configurations. We are also interested to explore general computational capabilities of DH2 architecture and to determine a set of problems that could be solved using it.

4 Personnel Supported

2000-2001:
  Dr. Wei-Min Shen (10%, 12/2000 – 08/2001)
  Dr. Sattiraju Prabhakar (100%, 12/2000 – 08/2001)
  Dr. Leila Meshkat (50%, 06/2001 – 08/2001)
  Aseem Mohanty (50%, 01/2001 – 06/2001)
2001-2002:
  Dr. Wei-Min Shen (15%, 09/01/2001 – 08/31/2002)
  Dr. Sattiraju Prabhakar (100%, 09/01/2001 – 03/31/2002)
2002-2003:
  Dr. Wei-Min Shen (15%, 09/01/2002 – 08/31/2003)
  Behnam Salemi (50%, 09/01/2002 – 08/31/2003)
2003-2004:
  Dr. Wei-Min Shen (35%, 09/01/2003 – 08/31/2004)
  Behnam Salemi (50%, 09/01/2003 – 12/31/2003)
  Maks Krivokon (50%, 01/01/2003 – 8/31/2004)
  Michael Rubenstein (25%, 05/01/2004 – 8/31/2004)

5 Publications

- B. Salemi, WM. Shen and P. Will. Hormone Controlled Metamorphic Robots, in the Proceedings of International Conference on Robotics and Automation, Seoul, Korea, May. 2001.
- WM. Shen, B. Salemi and P. Will. Hormone for self-reconfigurable robots, in the Proceedings of International Conference on Intelligent Autonomous Systems, IOS Press, pp. 918-925, 2000.
- WM. Shen, Y. Lu and P. Will, Hormone-based control for self-reconfigurable robots, in the Proceedings of International Conference on Autonomous Agents, Barcelona, Spain, 2000.
- WM. Shen, B. Salemi, and P. Will, Hormone-based Communication and Cooperation in Metamorphic Robots, submitted to the IEEE Transactions on Robotics and Automation, 2001.

- Shen, W.-M., C.-M. Choung, P. Will, Simulating Self-Organization for Multi-Robot Systems, *International Conference on Intelligent and Robotic Systems*, Switzerland, 2002.
- Shen, W.-M., B. Salemi, and P. Will, Hormone-Inspired Adaptive Communication and Distributed Control for CONRO Self-Reconfigurable Robots, *IEEE Transactions on Robotics and Automation*, (in print), October, 2002.
- Shen, W.-M. and B. Salemi, Distributed and Dynamic Task Reallocations in Robot Organization , IEEE Conference on Robotics and Automation, Washington DC, 2002.
- K. Støy, W.-M. Shen, and P. Will, "On the Use of Sensors in Self-Reconfigurable Robots." *In proceedings of the 7th international conference on simulation of adaptive behavior (SAB02)*, Edinburgh, UK, August 4-9, 2002.

- K. Støy, W.-M. Shen, and P. Will, "How to Make a Self-Reconfigurable Robot Run", *In proceedings of the 1st international joint conference on autonomous agents and multiagent systems (AAMAS'02)*, Bologna, Italy, July 15-19, 2002.
- K. Støy, W.-M. Shen, and P. Will, "Global Locomotion from Local Interaction in Self-Reconfigurable Robots", *In proceedings of the 7th international conference on intelligent autonomous systems (IAS-7)*, Marina del Rey, California, USA, March 25-27, 2002.
- Salemi, B., P. Will, and W.-M. Shen, Distributed Task Negotiation in Modular Robots, Special Issue on "Modular Robotics", *Journal of the Robotics Society of Japan* (RSJ), 2003.
- Salemi, B., Experimental Evaluation of Distributed Control for Chain-Type Self-Reconfigurable Robots, PhD Thesis, University of Southern California, 2003.
- Shen, W.-M., B. Salemi, and P. Will. Hormone-Inspired Adaptive Communication and Distributed Control for CONRO Self-Reconfigurable Robots, IEEE Transactions on Robotics and Automation, 18(5), October, 2002
- Shen, W.-M., P. Will, C.-M. Chuong, Self-organization and Distributed Control for Massive Robot Swarms, *Autonomous Robots*, (submitted), 2003
- Shen, W.-M., Self-Organization through Digital Hormones (invited), *IEEE Intelligent Systems*, 81-83, 8/2003.
- Stoy, K., W.-M. Shen, P. Will, Global Locomotion from Local Interaction in Self-Reconfigurable Robots, *Robotics and Autonomous Systems*, (in press) 2003.
- Shen, W.-M., P. Will, B. Khoshnevis, Autonomous Docking in Self-Reconfigurable Robots, *IEEE Transactions on Mechatronics*, (accepted) 2003.
- Shen, W.-M., P. Will, B. Khoshnevis, Self-Assembly in Space via Self-Reconfigurable Robots, International Conference on Robotics and Automation, Taiwan, 2003.
- Stoy, K., W.-M. Shen, P. Will, Implementing Configuration Dependent Gaits in Self-Reconfigurable Robots, International Conference on Robotics and Automation, Taiwan, 2003.
- B. Khoshnevis, P. Will, W.-M. Shen, Highly Compliant and Self-Tightening Docking Modules for Precise and Fast Connection of Self-Reconfigurable Robots, International Conference on Robotics and Automation, Taiwan, 2003.
- Shen, W.-M, P. Will, A. Galstyan, C.-M. Chuong, Hormone-inspired self-organization and distributed control of robotic swarms, *Autonomous Robots*, 17:93-105, 2004.
- Jiang, T-X. , Wideltz, RB., Shen, W.-M., Will, P., Wu, DY., Lin, CM., Jung, JS., Chuong, CM., 2004. Integument pattern formation involves genetic and epigenetic controls operated at different levels: Feather arrays simulated by a digital hormone model. *International Journal on Developmental Biology*, 48 (pages), 2004.
- Modi, P. J., W.-M. Shen, M. Tambe, and M. Yokoo, ADOPT: Asynchronous Distributed Constraint Optimization with Quality Guarantees, Artificial Intelligence, 2004.
- Salemi B., Will P., and Shen W.-M. Distributed Task Negotiation in Modular Robots, *Robotics Society of Japan, Special Issue on "Modular Robots"*, 2003.
- Shen, W.-M., Self-Organization through Digital Hormones, *IEEE Intelligent Systems*, July/August, 2003, pp 81-83.
- Shen, W.-M., B. Salemi, and P. Will, Hormone-Inspired Adaptive Communication and Distributed Control for CONRO Self-Reconfigurable Robots, *IEEE Transactions on Robotics and Automation*, 18(5), October, 2002.

- Stoy, K, W.-M. Shen, P. Will, Using Role-Based Control to Produce Locomotion in Chain-type Self-Reconfigurable Robots, IEEE Transactions on Mechatronics, 7(4), 410-417, Dec. 2002.
- Rubenstein, M., K. Payne, P. Will, W.-M. Shen, Docking among Independent and Autonomous CONRO Self-Recofigurable Robots, International Conference on Robotics and Automation. April-May 2004, New Orleans, USA.
- Salemi, B. and Wei-Min Shen. Distributed Behavior Collaboration for Self-Reconfigurable Robots. International Conference on Robotics and Automation. April-May 2004, New Orleans, USA.
- Behnam Salemi, Peter Will, Wei-Min Shen, Distributed Task Negotiation in Self-Reconfigurable Robots, International Conference on Intelligent Robots and Systems. Las Vegas, October 2003.
- Shen, W.-M., P. Will, B. Khoshnevis, Self-Assembly in Space via Self-Reconfigurable Robots, International Conference on Robotics and Automation, Taiwan, 2003.
- Stoy, K., W.-M. Shen, P. Will, Implementing Configuration Dependent Gaits in Self-Reconfigurable Robots, International Conference on Robotics and Automation, Taiwan, 2003.
- B. Khoshnevis, P. Will, W.-M. Shen, Highly Compliant and Self-Tightening Docking Modules for Precise and Fast Connection of Self-Reconfigurable Robots, International Conference on Robotics and Automation, Taiwan, 2003.

6 Interactions and Transitions

In 2000-2001, we presented our papers on self-organization at three international conferences: the international conference on Robotics and Automation, the international conference on Autonomous Agents, and the international conference on Intelligent Autonomous Systems.

During the lifetime of the project, Dr. Wei-Min Shen has been invited to give many talks:

- An invited plenary talk, the TTI/Vanguard Conference on the Future of Software, 2001.
- NASA Workshop on Human and Robotic Space Exploration, NASA Langley Research Center, November, 2001.
- NASA Ames Research Center, Hormone-Inspired Control for Self-Reconfigurable Robots, September, 2001.
- Naval Research Laboratory, Invited AI Seminar on self-reconfigurable robots, Washington, DC, June, 2002.
- UCLA CS Seminar, Self-Reconfigurable Robots and Digital Hormones, Los Angeles, January, 2002.
- Invited to Australian Center for Field Robotics Seminar, University of Sydney, Self-Reconfigurable Robots, 7/24/2003
- Invited to UC San Diego AI Seminar, Self-Reconfigurable Robots and Digital Hormones 2/22/2003.
-

In 2004, Dr. Wei-Min Shen served as a guest editor for the Special Issue for Self-Reconfigurable Robots, *IEEE Transactions on Mechatronics,* 2004.

7 New Discoveries, Inventions, or Patent Disclosures

US Patent **Award #6636781:** Distributed Control and Coordination of Autonomous Agents in a Dynamic, Reconfigurable System, October 21, 2003 by Shen, W.-M., B. Salemi, and P. Will.

8 Honors and Awards

This project has been received the following honors and awards:

- Just What We Need – Hormonal Robots, By Matthew Nelson, InformationWeek, Nov 28, 2000.
- Digital Hormones for Robots, SCIENCE AND ENGINEERING NEWS, HPCwire, www.newscientist.com, April 13, 2001.
- Digital Hormones for Robots, ***New Scientist,*** April 14, 2001 (p.22)
- Leading Research in Self-Reconfigurable Robots, The World Daily Newspaper, March, 2001.
- **The Best Paper Award:** The 7th International Conference on Simulation of Adaptive Behaviors (SAB2002) the Paper with Most Philosophical Consequences: On the Use of Sensors in Self-Reconfigurable Robots , by K. Støy and W.-M. Shen and P. Will.

During the 2002 Annual Conference of Artificial Intelligence, several Canadian newspapers reported the project. They are "Edmonton Journal" July 31, 2002, and "Calgary Herald" July 31, 2002.

In 2003, Dr. Wei-Min Shen has received a Phi Kappa Phi Faculty Recognition Award from the University of Southern California.

In 2004, Dr. Wei-Min Shen was invited to give a plenary talk at the 2004 International Conference on Complex Systems, Boston, (May 2004). Nature and Science had both reported the work from this project as follows:

nature.com
the world's best science on your desktop (May 28th, 2004)

Puckish Robots Pull Together: A story about prototype robots which practice docking maneuvers on an air hockey table was explained by Wei-Min Shen of USC's Viterbi School of Engineering. Given the hazards of human space travel, Shen believes robots are the best bet for building structures in space. "Assembly performed by astronauts would be too expensive and risky," he said. Shen's group is collaborating with NASA to develop intelligent robotic systems that can coordinate their own activities, requiring less human control and monitoring.

Science (August 8th, 2003): The work of USC School of Engineering robotics experts Wei-Min Shen and Peter Will is described in detail in a long report in the August 8 issue on "Shape Shifting Robots." Will and Shen's CONRO robots "have an especially spectacular ability to adapt on the fly.... 'The surprise in people's eyes when they see this is amazing,' Will says. 'When the thing gets up and walks all your human feelings about robots come out. Some people cheer for it, other people find it scary.' CONRO's adaptability comes from an innovative, decentralized control system, analogous to biological hormones. 'In the body, the same signal causes your hand to wave, your mouth to open and your legs to move,' explains Shen. Similarly in a CONRO robot, a module's reaction to signals (or 'hormones') from other modules depends on its current function....In the future, modular robots may be used to build power stations in space (a project the CONRO team is working on) or conduct search and rescue operations.